**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

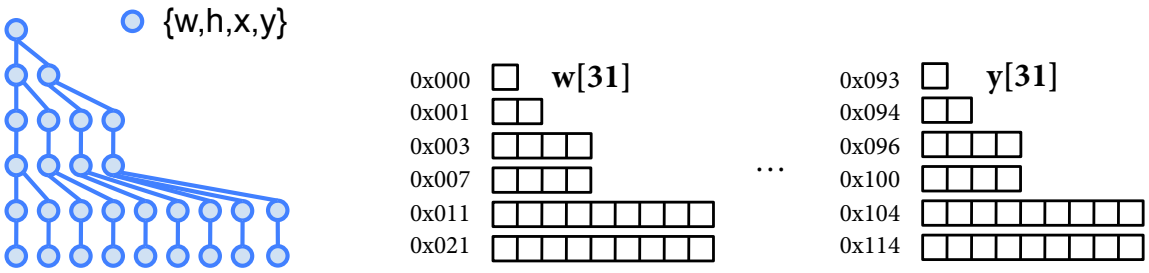
The common baseline for our two SIMD optimizations is to implement parallel preorder and postorder tree traversals as level-synchronous breadth-first parallel tree traversals. Reps first suggested such an approach to parallel attribute grammar evaluation [[CITE]], but did not implement it. Performance 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 in a way that avoids instruction divergence. Our insight is that we can cluster tasks (nodes) based on node attributes that influence control flow. We match the data layout to the new schedule, and optimize the clustering process to prevent the planning overhead to outweigh its benefit. The overall optimization can be thought of an extension to loop unswitching where the predicate is input-dependent and a sorting prepass guarantees that subintervals will branch identically.

### 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

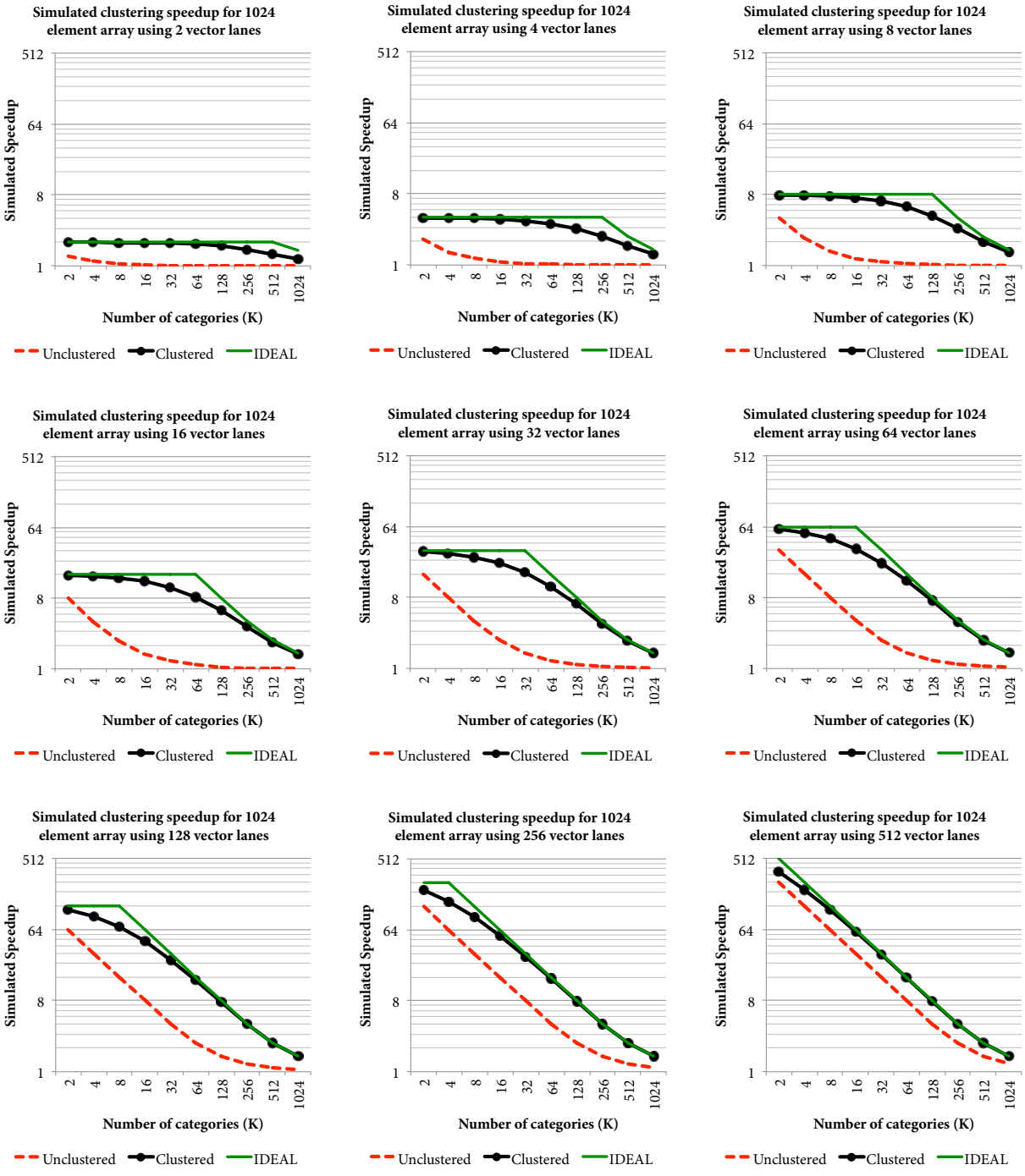



Figure 6.2: **blah**

```
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.**

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 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

```
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.**

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.

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.

### Nested Clustering

Clustering can also be used to address divergences induced by computations over neighboring nodes. They avoidable irregularities can take several forms:

- **Branches.** For the case of webpage layout, we saw cases where attributes of the parent node or children node influence instruction selection, such as whether to include a child node in a width computation. The properties can be included in the clustering condition to eliminate the corresponding instruction divergences.
- **Load imbalance in loops.** One node may have no children while another may have many. If the layout computation involves a loop, SIMD evaluation will perform the two loops in lock-step. Thus, as the nodes have different amounts of children, the SIMD lanes devoted to the first child will not be utilized: this is a load balancing problem. The number of children can be included in the clustering condition to eliminate load imbalance.
- **Random memory access in loops.** A further issue with lock-step loops over child nodes is memory divergence. A breadth-first layout would provide strided memory access, but if each level is clustered, the locations of a node's children may be random without further aid. We found a *nested* solution where *subtrees* are assigned to clusters. Instead of just associating nodes of a level with a cluster, our algorithm then treats the nodes of a cluster as roots. It recursively expands a subtree such that all of the cluster nodes share it (with respect to the attributes influencing control flow). The data layout follows the nested clustering, so parallel memory accesses to the children of nodes will be coalesced.

Each of these clusterings introduce an invariant for a cluster for optimizing performance within that cluster. However, the clustering condition is more discriminating. Cluster sizes may decrease, which would significantly decrease performance if cluster size shrinks below vector length size. Our evaluation explores these options in practice.

## 6.5 Automatically Staged SIMD Memory Allocation for Rendering

```
float *drawCircle (float x, float y, float radius) {
        float *buffer = malloc( (2 * sizeof(float) ) * round(radius))
        for (int i = 0; i < round(radius); i++) {
                buffer[2 * i] = x + cos(i * PI/radius);
                buffer[2 * i + i] = y + sin(i * PI/radius);
        }
        return buffer;
}
```

(a) **Naıve drawing primitive .**

```
int allocCircle (float x, float y, float radius) {
        return round(radius);
}
```

(b) **Allocation phase of drawing**.

```
int fillCircle(float x, float y, float radius, float *buffer) {
        for (int i = 0; i < round(radius); i++) {
                buffer[2 * i] = x + cos(i * PI/radius);
                buffer[2 * i + i] = y + sin(i * PI/radius);
        }
        return 0;
}
```

(c) **Tessellation phase of drawing**.

Figure 6.5: **Partitioning of a library function that uses dynamic memory allocation into parallelizable stages.**

## Problem

Dynamic memory allocation provides significant flexibility for a language, but it is unclear how to perform it on a GPU without significant performance penalties. This needed ended up leading to both performance and programmability issues in our design of a tessellation library that connects our GPU layout engine to our GPU rendering engine. Our insight is that the memory allocation may be staged using a variant of prefix sum node labeling. One pass gathers memory requests, a bulk allocation for the total amount is made, and then a scatter pass provides each node with a contiguous memory segment of it. We found manipulating memory addresses in this way to be error-prone, so we show how to use our synthesizer to automatically schedule use of the parallel memory allocator. Furthermore, we show how to syntactically hide the use of our allocation scheme through a macro that automatically expands into staged dynamic memory allocation and consumption calls.

For example, we found parallel dynamic memory allocation to simplify the transition between layout and rendering. All nodes that render a circle will call some form of drawCircle in Figure 6.5a. Depending on the size of the circle, which is computed as part of the layout traversals, a different amount of memory will be allocated. Once the memory is allocated, vertices will be filled in with the correct position. The rendering engine will then connect the vertices with lines and paint them to the screen. The processing of converting the abstract shape into renderable vertices is known as tessellation. We want our system to tesselate the

```
CBOX → BOX1 BOX2
{
        ...
        CBOX.render =
                drawCircle(CBOX.x, CBOX.y, CBOX.radius)
                + drawCircle(CBOX.x + 10, CBOX.y + 10, CBOX.radius * 0.5);
}
```

(a) **Call into inefficient library.**

```
CBOX → BOX1 BOX2
{
        ...
        CBOX.sizeSelf =
                allocCircle(CBOX.x, CBOX.y, CBOX.radius)
                + allocCircle(CBOX.x + 10, CBOX.y + 10, CBOX.radius * 0.5);
        CBOX.size = CBOX.sizeSelf +BOX1.size + BOX2.size;
        BOX1.buffer = CBOX.buffer + CBOX.sizeSelf;
        BOX2.buffer = BOX1.buffer + BOX1.size;
        CBOX.render =
                fillCircle(CBOX.x, CBOX.y, CBOX.radius, CBOX.buffer)
                + fillCircle(CBOX.x + 10, CBOX.y + 10, CBOX.radius * 0.5,
                        CBOX.buffer + allocCircle(CBOX.x, CBOX.y, CBOX.radius));
}
```

(b) **Macro-expanded calls into staged library**.

```
CBOX → BOX1 BOX2
{
        ...
        CBOX.render =
                @Circle(CBOX.x, CBOX.y, CBOX.radius)
                + @Circle(CBOX.x + 10, CBOX.y + 10, CBOX.radius * 0.5);
}
```

(c) **Sugared calls into staged library**.

Figure 6.6: **Use of dynamic memory allocation in a grammar for rendering two circles.**

display objects for each node in parallel.

## Staged Parallel Memory Allocation

We stage the use of dynamic memory into four logical phases:

1. Parallel request (bottom-up tree traversal to gather )
2. Physical memory allocation
3. Parallel response (top-down tree traversal to scatter)
4. Computations that consume dynamic memory (normal parallel tree traversals)

The staging allows us to parallelize the request and response stages. We reuse the parallel tree traversals for them, as well as for the actual consumption. The actual allocation of physical memory in stage 2 is fast because it is a single call.

Library functions that requires dynamic memory allocation are manually rewritten into allocation request (Figure 6.5b) and memory consumption fragments (Figure **??**). The transformation was not onerous to perform on our library primitives and, in the future, might be automated.

Invocations of the original in the attribute grammar are rewritten to use the new primitives. For example, drawing two circles (Figure 6.6a) is split into calls for allocation requests, buffer pointer manipulation, and buffer usage (Figure 6.6b). The transformation increases memory consumption costs due to book keeping of allocation sizes.

The result of our staging is three logical parallel passes, which, in practice, is merged into two parallel passes over the tree. The first pass is bottom up, similar to a prefix sum: each node computes its allocation requirements, adds that to the allocation requirements of its children,and then the process repeats for the next level of the tree. The sizeSelf and *size* attributes are used for the first pass. Once the cumulative memory needs is computed, a bulk memory allocation occurs, and then a parallel top-down traversal assigns each node a memory span from buffer to buffer + selfSize. Finally, the memory can be used for actual computations through normal parallel passes. Memory use can occur immediately upon computation of the buffer index, so the last two logical stages are merged in implementation.

## Automation with Automatic Scheduling and Macros

Manually manipulating the allocation requests and buffer pointers is error prone. We eliminated the problem through two automation techniques: automatic scheduling to enforce correct parallelization and macro expansion to encapsulate buffer manipulation.

To enforce proper parallelization, we relied upon our synthesizer to schedule the calls. If the synthesizer cannot schedule allocation calls and buffer propagation, it reports an error. Our insight is that, implicit to our staged representation, we could faithfully abstract the memory manipulations as foreign function calls. Our synthesizer simply performs its usual scheduling procedure.

To encapsulate buffer manipulation, we introduced the macro '@'. Code that uses it is similar to code that assumes dynamic memory allocation primitives: the slight syntactic difference can be seen between Figure 6.6c and Figure 6.6a. Our macros (implemented in OMetaJS [[CITE]]) automatically expand into the form seen in Figure 6.6b.

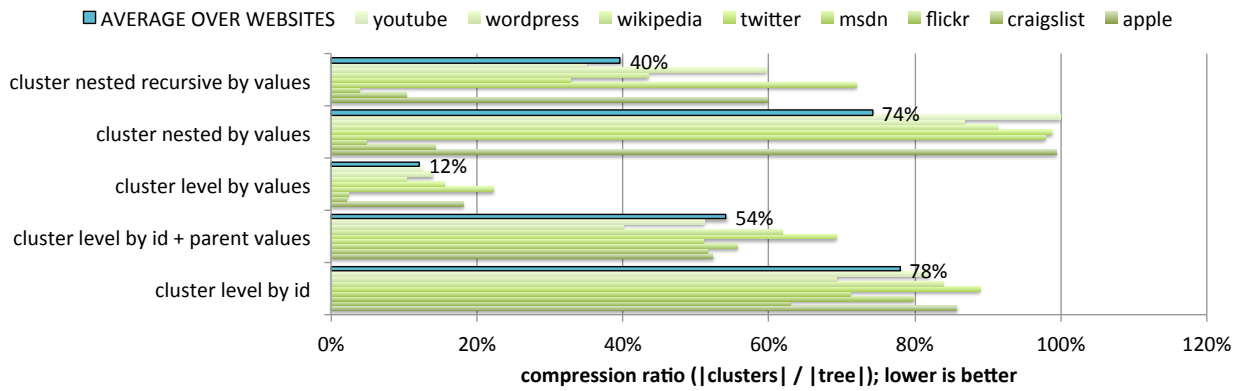



Figure 6.7: **Compression ratio for different CSS clusterings.** Bars depict compression ratio (number of clusters over number of nodes). Recursive clustering is for the reduce pattern, level-only for the map pattern. ID is an identifier set by the C3 browser for nodes sharing the same style parse information while value is by clustering on actual style field values.

## 6.6 Evaluation

### SIMD Clustering

We evaluate several aspects of our clustering approach. First, we examine applicability to various visualizations. Second, we evaluate the speed and performance benefit. Clustering provides invariants that benefit more than just vectorization, so we distinguish sequential vs. parallel speedups. Finally, there are different options in what clusters to form, so for each stage of evaluation, we compare impact.

#### Applicability

We examined idealized speedup for several workloads:

- **Synthetic.** For a controlled synthetic benchmark, we simulated the effect of increasing number of clusters on speedup for various SIMD architectures. Our simulation assumes perfect speedups for SIMD evaluation of nodes run together on a SIMD unit. The ideal speedup is a function of the minimum of the SIMD unit's length (for longer clusters, multiple SIMD invocations are mandatory) and the number of clusters (at least one SIMD step is necessary for each cluster). Figure 6.2 shows, for architectures of different vector length, that the simulated speedup from clustering (solid black line with circles) is close to the ideal speedup (solid green line).

- **Data visualization.** For our data visualizations, we found that, across the board, all of the nodes of a level shared the same type. For example, our visualization for

multiple line graphs puts the root node on the first level, the axis for each line graph on the second level, and all of the actual line segments on the third level.

- **CSS.** We analyzed potential speedup on webpages. Webpages are a challenging case because an individual webpage features high visual diversity, with popular sites using an average of 27KB of style data per page. [1]. We picked 10 popular websites from the Alexa Top 100 US websites that rendered sufficiently correctly in the C3 [[CITE]] web browser. It was also challenging in practice because it required clustering based on individual node attributes, not just the node type.

  Figure fig:csscompression compares how well nodes of a webpage can be clustered. It reports the *compression ratio*, which divides the number of clusters by the number of nodes. Sequential execution would assign each node to its own cluster, so the ratio would be 1. In contrast, if the tree is actually a list of 100 elements, and the list can be split into 25 clusters, the ratio would be 25%. Assuming infinite-length vector processors and constant-time evaluation of a node, the compression ratio is the exact inverse of the speedup. A ratio of 1 leads to a 1X speedup, and a compression ratio of 25% leads to a 4X speedup.

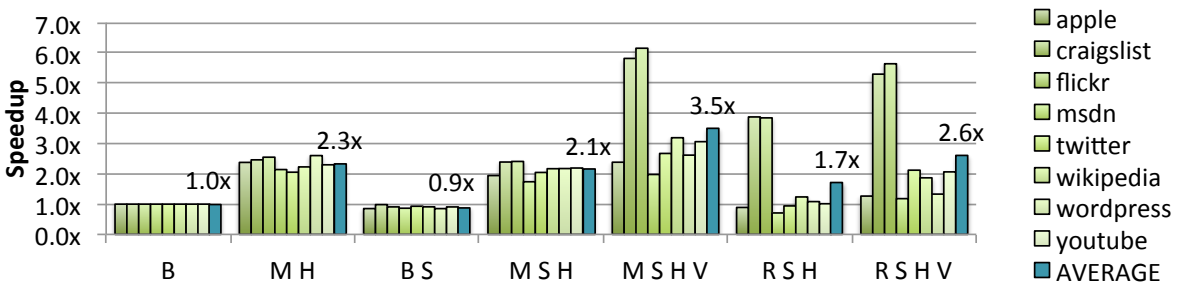
  Clustering each level by attributes that influence control flow achieved a 12% compression ratio (Figure fig:csscompression): an 8.3X idealized speedup. When we strengthened the clustering condition to enforce stronger invariants in the cluster, such as to consider properties of the parent node, the ratio quickly worsened. Thus, we see that our basic approach is promising for websites on modern subword-SIMD instruction sets, such as a 4-wide SSE (x86) and NEON (ARM), and the more recent 8-wide AVX (x86). Even longer vector lengths are still beneficial because some clusters were long. However, eliminating all divergences requires addressing control flows influenced by attributes of node neighbors, which leads to poor compression ratios. Thus, we emphasize that 8.3X is an upper bound on the idealized speedup: not all branches in a cluster are addressed.

Empirically, we see that clustering is applicable to CSS, and in the case of our data visualizations, unnecessary. Vectorization limit studies based on analyzing dynamic data dependencies from program traces suggest that general programs can be much more aggressively vectorized, so clustering may be the beginning of one such approach [[CITE]].

### Speedup

We evaluate the speedup benefits of clustering for webpage layout. We take care to distinguish sequential benefits from parallel, and of different clustering approaches. Our implementation was manual: we examine optimizing one pass of the C3 [[CITE]] browser's CSS layout engine that is responsible for computing intrinsic dimensions. The C3 browser was

[1] https://developers.google.com/speed/articles/web-metrics




B =breadth first, S = structure splitting, M = level clustering, R = nested clustering, H = hoisting, V = SSE 4.2

Figure 6.8: **Speedups from clustering on webpage layout.** Run on a 2.66GHz Intel Core i7 (GCC 4.5.3 with flags -O3 -combine -msse4.2) and does not preprocessing time.

written in C#, so we wrote our optimized traversal in C and pinned the memory for shared access. We use a breadth-first tree representation and schedule for our baseline, but note that doing such a layout already provides a speedup over C3's unoptimized global layout.

For our experimental setup, we evaluate the same popular webpages above that rendered legibly with the experimental C3 browser. Benchmarks ran on a 2.66GHz Intel Core i7 (GCC 4.5.3 with flags -O3 -combine -msse4.2). We performed 1,000 trials, and to avoid warm data cache effects, iterated through different webpages.

We first examine sequential performance. Converting an array-of-structures to a structure-of-arrays causes a 10% slowdown (B S in Figure 6.8). However, clustering each level and hoisting computations shared throughout a cluster led to a 2.1X sequential benefit (M S H). Nested clustering provided more optimization opportunities, but the compression ratio worsened: it only achieved a 1.7X sequential speedup (R S H). Clustering provides a significant sequential speedup.

Next, we examine the benefit of vectorization. SSE instructions provide 4-way SIMD parallelism. Vectorizing the nested clustering improves the speedup from 1.7X to 2.6X, and the level clustering from 2.1X to 3.5X. Thus, we see significant total speedups. The 1.7X relative speedup of vectorization, however, is still far from the 4X: level clustering suffers from randomly strided children, and the solution of nested clustering sacrifices the compression ratio.

### Power

Much of our motivation for parallelization is better performance-per-Watt, so we evaluate power efficiency. To measure power, we sampled the power performance counters during layout. Each measurement looped over the same webpage over 1s due to the low resolution of the counter. Our setup introduces warm cache effects, but we argue it is still reasonable because a full layout engine would use multiple passes and therefore also have a warm cache across traversals.

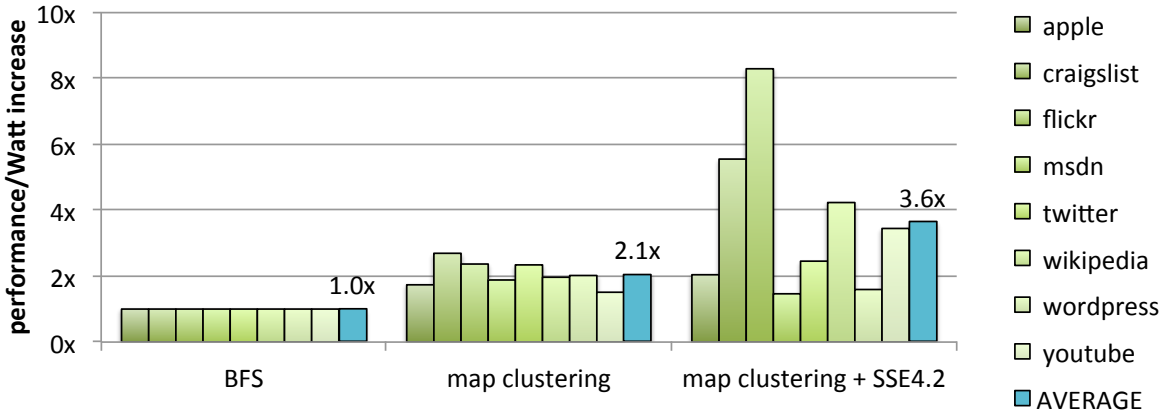



Figure 6.9: **Performance/Watt increase for clustered webpage layout.**

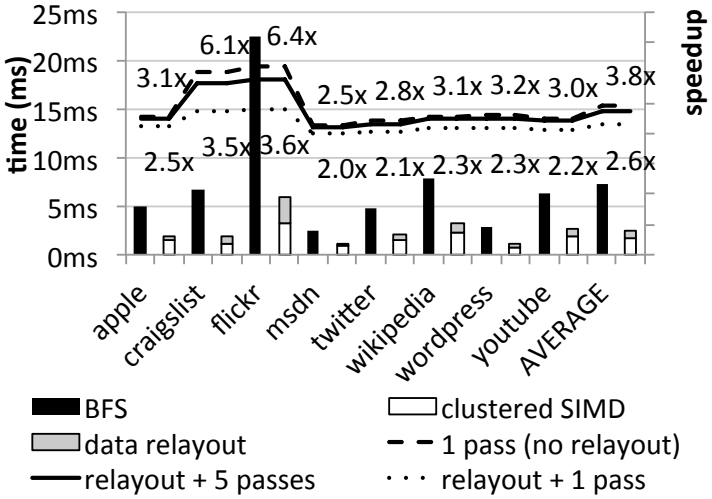



Figure 6.10: **Impact of data relayout time on total CSS speedup.** Bars depict layout pass times. Speedup lines show the impact of including clustering preprocessing time.

In Figure 6.9, we show a 2.1X improvement in power efficiency for clustered sequential evaluation, which matches the 2.1X sequential speedup of Figure 6.8. Likewise, we report a 3.6X cumulative improvement in power efficiency when vectorization is included, which is close to the 3.5X speedup. Thus, both in sequential and parallel contexts, clustering improves performance per Watt. Furthermore, it supports the general reasoning in parallel computing of 'race-to-halt' as a strategy for improving power efficiency.

### Overhead

Our final examination of clustering is of the overhead. Time spent clustering before layout must not outweigh the performance benefit; it is an instance of the planning problem.

For the case of data visualization, we convert the data structure into arrays with an offline preprocessor. Thus, our data visualizations experience no clustering cost.

For webpage layout, clustering is performed on the client when the webpage is received. We measured performing sequential two-pass clustering. Figure 6.10 shows the overhead relative to one pass using the bars. The highest relative overhead was for the Flickr homepage, where it reaches almost half the time of one pass. However, layout occurs in multiple passes. For a 5-pass layout engine where we model each pass as similar to the one we optimized, the overhead is amortized. The small gap between the solid and dashed lines in Figure 6.10 show there is little difference when we include the preprocessing overhead in the speedup calculation.

## Staged SIMD Memory Allocation

We evaluate three dimensions of our staged memory allocation approach: flexibility, productivity, and performance. First, it needs to be able to express the rendering tasks that we encounter in GPU data visualization. Second, it should some form of productivity benefit for these tasks. Finally, the performance on those tasks must be fast enough to support real-time animations and interactions of big data sets.

### Flexibility

Our staged structuring and automation approach cannot express all dynamic memory usage patterns, so it is important to validate that it works on common patterns that occur in visualization. We found the three following patterns to be important:

- **Functional graphics.** Functional graphics primitives used in languages such as Scheme, O'Caml, and Haskell follow the form that we used for Circle. For example, many of our visualizations use simple variants such as 2D rectangles, 3D line, and arcs.

- **Linked view**. Multiple renderable objects can be associated with one node, which we can use for providing different views of the same data. Such functionality is common for statistical analysis software:

```
render :=  @Circle(x,y,r)  + @Circle(offsetX + abs(x), offsetY + abs(y), r);
```

- **Zooming.** We can use the same multiple representation capability for a live zoomed out view ("picture-in-picture"):

```
render :=
  @Circle(x, y, radius)
   + @Circle(xFrame + x*zoom, yFrame + y*zoom, radius *zoom);
```

- **Visibility toggles.** Our macros support conditional expressions, which enables controlling whether to render an object. For example, a boolean input attribute can control whether to show a circle: `render := isOn ? @Circle(0,0,10) : 0;`
- **Alternative representations.** Conditional expressions also enable choosing between multiple representations, not just on/off visibility:

```
render :=
  isOff ? 0
    : mouseHover ? @CircleOutline(0,0,10)
    : @Circle(0,0,10,5) ;
```

### Productivity

Productivity is difficult to measure. Before using the automation extensions for rendering, we repeatedly encountered bugs in manipulating the allocation calls and memory buffers. The bugs related both to incorrect scheduling and to incorrect pointer arithmetic. Our new design eliminates the possibility of both bugs.

One suggestive productivity measure is of how many lines of code the macro abstraction eliminates from our visualizations. We measured the impact on using it for 3 of our visualizations. The first visualization is our HBox language extended with rendering calls, while the other two are interactive reimplementations of popular visualizations: a treemap [[CITE]] and multiple 3D line graphs [[CITE]].

Table 6.1: Lines of Code Before/After Invoking the '@' Macro

| Visualization | Before (loc) | After (loc) | Decrease |
|---|---|---|---|
| HBox | 97 | 54 | 44% |
| Treemap | 296 | 241 | 19% |
| GE | 337 | 269 | 20% |

Table 6.1 compares the lines of code in visualizations before and after we added the macros. Using the macros eliminated 19–44% of the code. Note that we are *not* measuring the macro-expanded code, but code that a human wrote.

As shown in Figure 6.6, the eliminated code is code that was introduced by staging the library calls. Porting unstaged functional graphics calls to the library, is in practice, an alpha renaming of function names. Using the '@' macro eliminates 19–44% of the code that would have otherwise been introduced and completely eliminates two classes of bugs (scheduling and pointer arithmetic), so the productivity benefit is non-trivial.

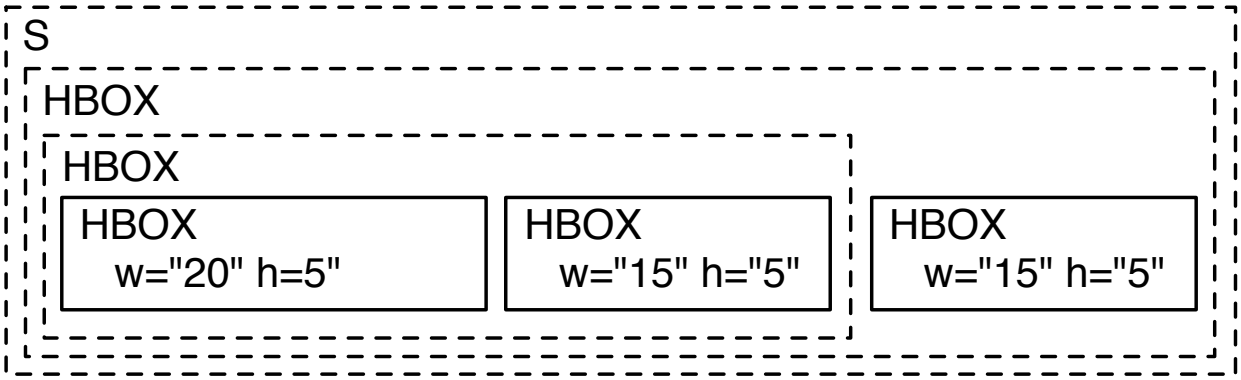
**Performance**

## 6.7 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. sims limit studies
3. duane
4. trishul
5. 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 = $\text{input}_w()$; HBOX.h = $\text{input}_h()$ }

$HBOX_0 \rightarrow HBOX_1\ HBOX_2$
{ $\text{HBOX}_1$.x = $\text{HBOX}_0$.x;
$\text{HBOX}_2$.x = $\text{HBOX}_0$.x + $\text{HBOX}_1$.w;
$\text{HBOX}_1$.y = $\text{HBOX}_0$.y;
$\text{HBOX}_2$.y = $\text{HBOX}_0$.y;
$\text{HBOX}_0$.h = max($\text{HBOX}_1$.h, $\text{HBOX}_2$.h);
$\text{HBOX}_0$.w = $\text{HBOX}_1$.w + $\text{HBOX}_2$.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).

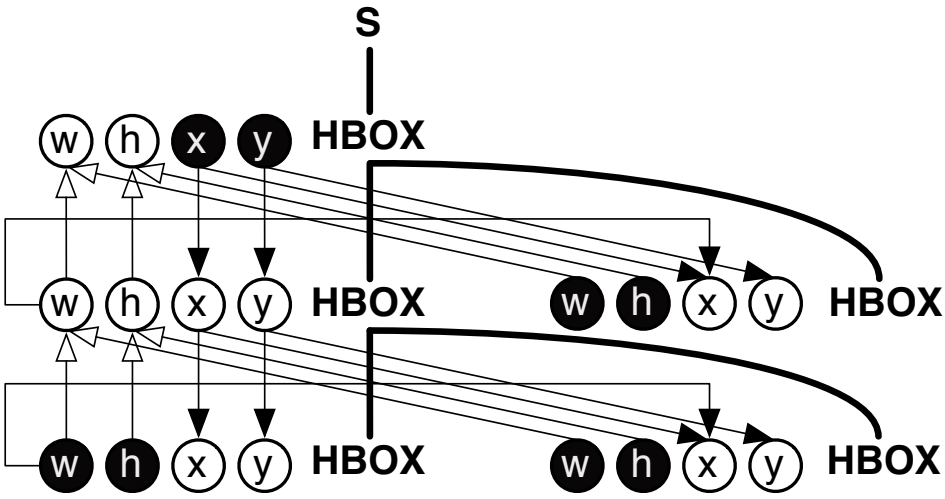



Figure 7.2: **Data dependencies.** Shown for constraint tree in Figure 2 (a). Circles denote attributes, with black circles being input() sources. Thin lines show data dependencies and thick lines show production derivations.

```
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)

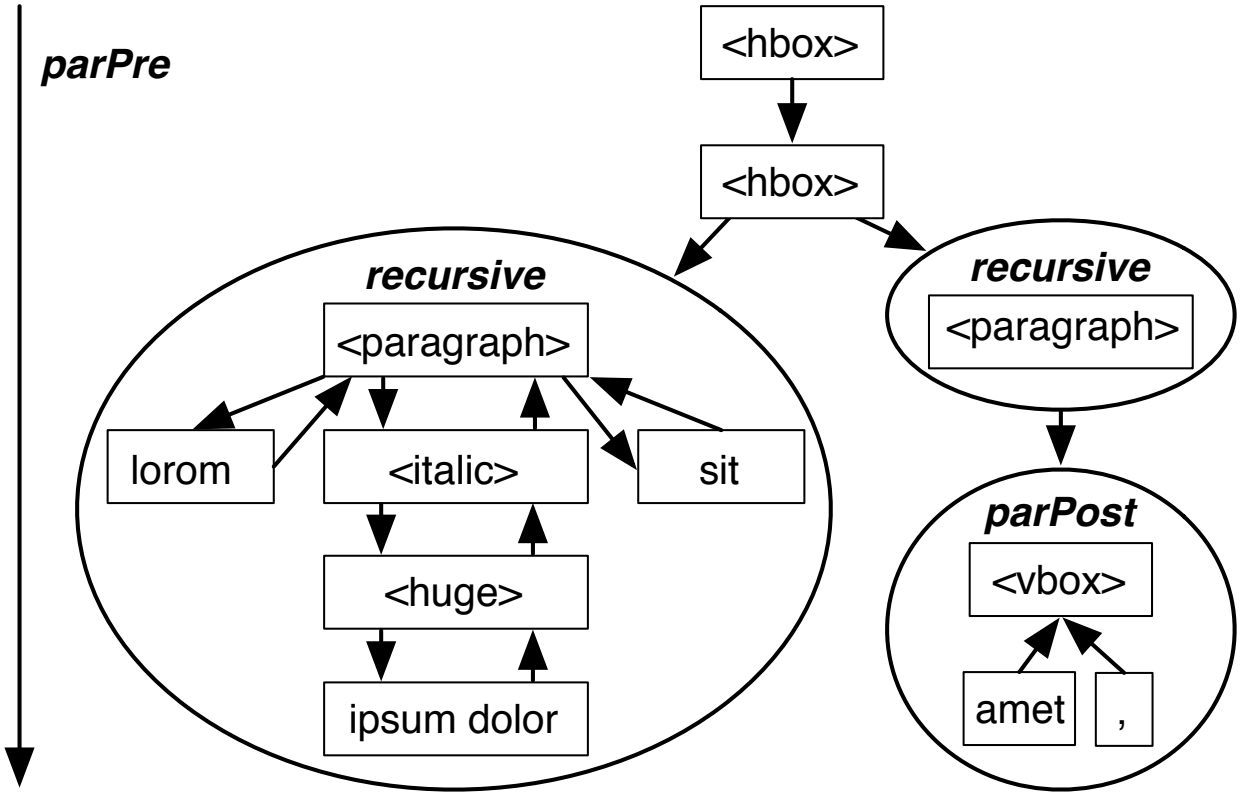



Figure 7.4: **Nested traversal for line breaking**. The two paragraph are traversed in parallel as part of a preorder traversal and a sequential recursive traversal is used for words within a paragraph.

# Appendix A

# Layout Grammars