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.

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

We start by examining challenges for building layout languages and our high-level solution of automation through attribute grammars. Throughout this and the remaining chapters, we focus on the design and implementation of one simple layout widget. We will show how our support of it generalizes to common layout languages and, more generally, computations over trees.

Important properties for layout languages and others

Layout languages are some of the most common – for one gauge, there are over 634 million websites live in 2012, with 51 million added that year [1]. Beyond the CSS and HTML languages used for webpage layout, designers also use LaTeX [[CITE]] for document layout, D3 [[CITE]] for data visualization, Swing [[Swing]] for GUI layout, and even specialize within these domains such as by using markdown for text.

Popular layout languages foster designer productivity by providing abstractions that are rich and numerous. The alternative is analogous to asking a programmer to write in a low-level language such as assembly: designers should not manually specify, for each element, the position on a canvas and the style. Instead, layout languages resemble constraint systems where designers declare high-level properties. For example, the high-level program `hello world` states that the words `hello` and `world` should be rendered, and word `world` should follow line-wrapping rules for its positioning after `hello`. Layout languages may provide quite complicated constraints – for example, most document layout languages resort to defining their line wrapping rule in a flexible low-level language. Likewise, they may provide many featuress, such as in the 250+ pages of rules for the CSS language. Adding to the sophistication, many

[1] http://news.netcraft.com/archives/2012/12/04/december-2012-web-server-survey.html

languages support designers adding their own constraints, such as through macros in LaTeX, percentage constraints in CSS, and arbitrary functions in Adobe Flex [[CITE]].

The richness of popular layout languages comes at the cost of of complicating their design and implementation:

- **Safe semantics.** Does every input layout have exactly one unique rendering? Are the constraints restricted enough such that an efficient implementation is feasible for low-power devices, big data sets, and fast animation? When a feature is added, does it conflict with anything of the above properties? We want an automated way to verify such properties.

- **Safe implementation.** As a layout language grows in popularity, it grows in features. Likewise, developers will port it to many platforms and optimize it, and in cases such as CSS, reimplement it from scratch. Does the implementation conform to the intended semantics? Conformance bugs for CSS plague developers [[CITE]], and failures to match LaTeX's semantics have killed multiple attempts to modernize the implementation. We want an automated way to ensure that the implementation matches the specification.

- **Advanced implementation.** Layout languages tend to add feature as they evolve. However, the implementation of each feature also has demands that increase with time: improved speed and memory footprint, better debugging support, etc. Browser layout engines for CSS are currently over 100,000 lines of optimized C++ code, and most rich layout languages thus far have resisted parallelization. We want automation techniques to lower the implementation burden and more aggressively target those goals.

Our idea is to declaratively specify layout languages and automatically compile them into an efficient implementation. At runtime, an instance of layout will be processed through the previously generated layout engine (Figure 2.1). The compiler is responsible for checking the semantics of the layout features and, by construction, provides a correct implementation. Furthermore, instead of manually optimizing the code for every individual feature, language designers instead write generic compiler optimizations. As a similar implementation benefit, we automatically target multiple platforms for the same layout language, such as scripting languages in order to use their debuggers, and multicore and GPU languages to gain magnitudes of speedups.

We show that the attribute grammar formalism supports specification of layout languages. It is unclear how to encode complicated layout language features with the traditional formalism, so we support a rich form of attribute grammars and reduce reasoning about them to handling a more traditional formalism (reducer in Figure 2.1). The remainder of this chapter introduces the high-level attribute grammar formalism, how to specify layout languages using it, and an intuition for the reduction into a lower-level formalism.

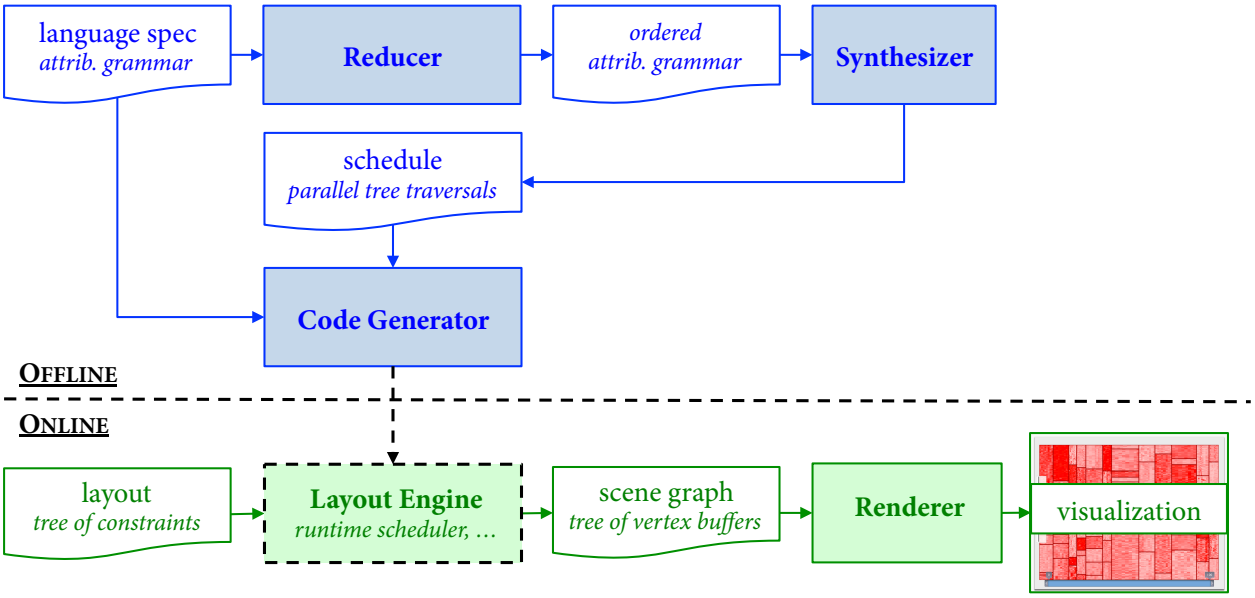


Figure 2.1: **Layout engine architecture.**

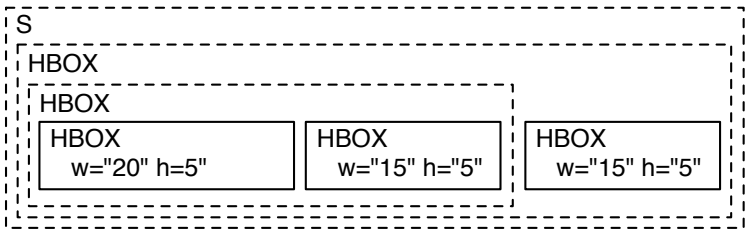
2.2 Background: Layout with Classical Attribute Grammar

This section describes specifying a simple layout language as an attribute grammar and two classical implementation strategies. We reuse the example throughout our work to explore various concepts.

Attribute Grammars

Consider solving the tree of horizontal boxes shown in Figure 2.2a. As input, a webpage author provides a tree with constraints (Figure **??**). Only some node attribute values are provided: in this case, only the widths and heights of leaf nodes. The meaning of a horizontal layout is that, as is visualized, the boxes will be placed side-by-side. The layout engine must solve for all remaining x, y, width, and height attributes.

We declaratively specify the layout language of horizontal boxes, `H-AG` , as shown in Figure 2.2c, with an attribute grammar [**oag**, **Meyerovich:2010**, **htmlag**]. First, the specification defines the set of well-formed input trees as the derivations of a context-free grammar. We use the standard notation [[CITE]]. In this case, a document is an unbalanced binary tree of arbitrary depth where the root node has label `S` and intermediate nodes have label `HBOX`. Second, the specification defines semantic functions that relate attributes associated with each node. For example, the width of an intermediate horizontal node is the sum of its children widths. Likewise, the width of a leaf node is provided by the user, which is encoded by the nullary function call $input_w()$:



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

```
1  <S>
2    <HBox name=child>
3      <HBox name=left>
4        <HBox name=left w=20 h=5/>
5        <HBox name=right w=15 h=5/>
6      </HBox>
7      <HBox name=right w=15 h=5/>
8    </HBox>
9  </S>
```

(b) **Textual encoding of input tree.**

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

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

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

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

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

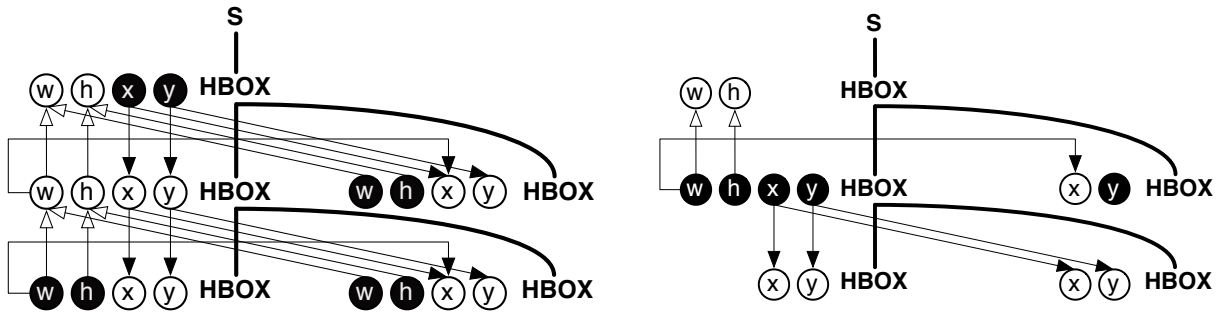
$Prod \rightarrow V \rightarrow V^*$

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

$Attrib \rightarrow id.id$

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

Figure 2.2: 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).



(a) Initial dependencies.

(b) After first round of evaluation.

Figure 2.3: **Dynamic data dependencies and evaluation**. Shown for constraint tree in Figure ZZZ (a). Circles denote attributes, with black circles denote attributes with resolved dependencies such as input()s. Thin lines show data dependencies and thick lines show production derivations. Second chart shows the dependency graph resulting from evaluating all source nodes and marking them as resolved.

$HBOX \rightarrow \epsilon$ { HBOX.w = $input_w()$; ... } /* leaf */

$HBOX_0 \rightarrow HBOX_1\ HBOX_2$ /* binary node */
{ ... HBOX_0.w = HBOX_1.w + HBOX_2.w }

The specification intentionally does not define the evaluation order. For example, the specification does not state whether to compute a node's width before its height. Likewise, our optimized approach will compute the attributes as a sequence of tree traversals, but the specification does not state what those traversals are. Leaving the evaluation order unspecified provides freedom for our compilers to pick an efficient parallel order. Irrespective of whatever evaluation order is ultimately used to solve for the attribute values, the statements define constraints that must hold over the computed result. Attribute grammars can therefore be thought of as a single assignment language where attributes are dataflow variables [[CITE]].

The language of attribute grammars is defined in Figure 2.2d. In addition the defining the context free grammar, it supports single-assignment constraints over attributes of nodes in a production. Our example uses the following encoding. Semantic functions are pure and left uninterpreted, so, for example, we encode the addition of widths as "HBOX_0.w = f(HBOX_1.w, HBOX_2.w)". Our program analysis techniques do not need to know the contents of the function, just that the output of a call depends purely on the inputs. For the same reason, we encode constant values as nullary function calls.

To specify grammars more complicated than `H-AG` , we describe linguistic extensions for richer functional specifications (Section 2.3) and, to control the evaluation order, behavioral specification (Chapters 3 and 4).

```
1  input: G = (V, E)
2  output: Map
3  Map ← ∅
4  E′ ← E
5  V′ ← V
6  for a ∈ V′ where ∄(n, a) ∈ E′:
7    Map ← Map ∪ {a → eval(a)}
8    V′ ← V′ − a
9    E′ ← E′ ({a} × V)
10 repeat until E′ = ∅
11 return Map
```

Figure 2.4: **Dynamic attribute grammar evaluator.** It selects attributes in a safe order by dynamically removing dependency edges as they are resolved.

Dynamic data dependencies and dynamic evaluation

A simple and classic evaluation strategy is to *dynamically* compute over a tree. The evaluator dynamically follows the dynamic data dependencies between instances of attributes. The dynamic evaluation strategy is too slow for our use cases, but it introduces the key concepts of dynamic data dependencies, the dynamic semantics of attributes grammars, and the corresponding dynamic interpreter.

An instance of a document corresponds to the dependency graph shown in Figure 2.3a. Each attribute of a tree node is either a source, meaning its value can be computed based on other known values, or it cannot be evaluated until other attribute values are known. It is a dynamic dependency graph in that each data dependency in the static code may be instantiated as multiple data dependencies given a tree at runtime.

The dynamic data dependency graph leads to a simple semantics for the result of evaluation. The graph corresponds to a system of equations where edges link instance variables. For example, static code `HBOX$_2$.x = HBOX$_0$.x + HBOX$_1$.w` instantiates twice for the Figure **??**: once for each x attribute with an incoming elbow connector. The value of both xs are constrained by distinct instances of the above constraint. If the dependency graph is a directed acyclic graph and each attribute appears on the lefthand side of exactly one equality statement (*dataflow variables*), there is exactly one solution to the system of equations.

A simple procedure solves an instance of a system of equations: topological traversal. The algorithm is as follows: The algorithm literately finds an attribute whose dependencies have all been previously resolved, evaluates the attribute, and repeats. If the input graph is a directed acyclic graph, this procedure is guaranteed to terminate. The insight is that a directed acyclic graph has at least one fringe node, the loop removes them, and removing these nodes yields a smaller directed acylic graph.

The dynamic evaluation strategy provides a small explanation for the natural semantics, but it leaves several challenges. First, runtime manipulation of a dynamic dependency graph introduces high overheads because every dynamic dependency edge must be manipulated. Second, it is unsafe. For example, a cycle in the dependency graph causes the above eval-

<Top> → *<Top>** | *<Trait>* | *<Class>* | *<Interface>*

<Interface> → **interface** *id* *<AttribDecl>**

<AttribDecl> → **var** *id* : *id*
| **input** *id* : *id*
| **input** *id* : **?** *id*
| **input** *id* : *id* = *val*

<Trait> → **trait** *id* *<Child>** *<AttribDecl>*<TopConstraint>**

<Class> → **class** *id* (*id**) : *id* *<Child>** *<AttribDecl>** *<TopConstraint>**

<Child> → *id* : (*id* | [*id*])

<TopConstraint> → *<Constraint>*
| **loop** *id* (*<Constraint>* | *<Lhs>* := **fold** *<Expr>* .. *<Expr>*)*

<Constraint> → *<Lhs>* := *<Expr>*

<Expr> → *<Rhs>* | *unop* *<Expr>* | *<Expr>* *binop* *<Expr>*

<Rhs> → *<Lhs>* | **self** *<Suffix>* . *id* | *id* *<Suffix>* . *id*

<Lhs> → *id* | *id* . *id*

<Suffix> → **$i** | **$-** | **$$**

Figure 2.5: **EBNF Syntax for key forms in the functional specification language.** We omit semicolons and other decorations; see the examples for more detailed forms.

uation strategy to get stuck, so dynamic evaluators must introduce runtime cycle check. Designers can build layout widgets that, depending on how they are invoked, fail to display!

2.3 Desugaring Loops and Other Modern Constructs

The attribute grammar formalism was invented for describing semantics [[CITE]] and before many modern constructs became mainstream: we had to design extensions for improved expressiveness and maintainability. Our extensions exploit concepts from structured, object-oriented, and functional programming. Other language designers have build such extensions as well [[CITE]]: our challenge was to make expressive extensions that facilitate effective parallelization and do not overly complicate language and tool implementation. This section documents the language features and how they simplify implementation, and leaves performance optimization to the next chapter.

Our key insight is that pre- and post-processing supports desugaring a feature-rich at-

tribute grammar into the canonical attribute grammar notation. Tools then operate at the most appropriate stage, such as our scheduler on the small, canonical attribute grammar representation. Likewise, our code generators take a generated schedule and relate it back to a representation from early in the preprocessing stage. Many of our features are built as explicit compiler stages, but over time, we found that declarative tree rewriting systems such as ANTLR and OMeta support automating individual stages.

The subsections below illustrate the various features and how they relate to attribute grammars. Figure 2.5 shows the syntax of our functional specification language. Section **??** shows the optional extension for behavioral specification and Section **??** for SIMD rendering macros.

Interfaces for Encoding Tree Grammars

Attribute grammars are an extension to the tree grammar formalism for defining input trees, so improving the abstraction capabilities of tree grammars also aids the ability to structure attribute grammars. In particular, we found the need to support abstracting over similar types of non-terminals. Our solution is to provide a notion of classes and interfaces. Our core extension is macro-expressible with attribute grammars and therefore reduces implementation requirements, though it is still important enough that it merits deeper compiler support.

Consider the code duplication performed when extending `H-AG` with vertical boxes. The children of a HBox could be a horizontal box or a vertical box, and the same for the children of a vertical box. Figure **??** shows that the 3 productions of `H-AG` grew to be 11. The example highlights that canonical attribute grammars cannot abstract over node types. Adding a new box type requires modifying all previous box classes, and in the presence multiple children, extension suffers exponential costs.

To abstract over node types, we introduce the notion of classes and interfaces (Figure 2.6b). Classes are similar to the productions of an attribute grammar: the class name specifies the production's lefthand side non-terminal and the children block specifies the production's righthand side. Unlike attribute grammars, an interface name is used for the righthand side rather than the class name. HBox and VBox implement interface BoxI, so any class specified to have a BoxI child can have a HBox or VBox child within the concrete tree.

Classes and interfaces are formally equivalent to tree grammars in the sense of a 1-to-1 correspondence between trees described by both. First, a tree grammar can be expressed with classes and interfaces by treating all productions with the same lefthand-side non-terminal as different classes belonging to the same interface. In the other direction, each interface can be expressed as a production that derives the classes, and the classes expand into productions. Figures 2.6b and 2.6c demonstrate the correspondence for `H-AG` . The induced implementation requirements are therefore slight in the sense that the construct is sugar for a pattern in the canonical attribute grammars.

We depart from the correspondence for the encoding of trees in two ways. First, we represent input as a tree with labeled nodes and edges. Node labels denote the class and

$$
\begin{aligned}
S &\rightarrow HBOX\ j\ VBOX \\
HBOX &\rightarrow \epsilon \\
HBOX_0 &\rightarrow HBOX_1\ HBOX_2 \\
HBOX_0 &\rightarrow VBOX_1\ HBOX_2 \\
HBOX_0 &\rightarrow HBOX_1\ VBOX_2 \\
HBOX_0 &\rightarrow VBOX_1\ VBOX_2 \\
VBOX &\rightarrow \epsilon \\
VBOX_0 &\rightarrow HBOX_1\ HBOX_2 \\
VBOX_0 &\rightarrow VBOX_1\ HBOX_2 \\
VBOX_0 &\rightarrow HBOX_1\ VBOX_2 \\
VBOX_0 &\rightarrow VBOX_1\ VBOX_2
\end{aligned}
$$

(a) **Canonical attribute grammar.**

```
1  interface BoxI { }
2  class HBoxLeaf : BoxI { }
3  class HBoxBinary : BoxI {
4    children {
5      left: BoxI;
6      right: BoxI;
7    }
8  }
9  class VBoxLeaf : BoxI { }
10 class VBoxBinary : BoxI {
11   children {
12     left: Box;
13     right: Box;
14   }
15 }
```

(b) **Interface sugar.**

$$
\begin{aligned}
S &\rightarrow BOX \\
BOX &\rightarrow HBOX\ j\ VBOX \\
HBOX &\rightarrow \epsilon \\
HBOX_0 &\rightarrow BOX_1\ BOX_2 \\
VBOX &\rightarrow \epsilon \\
VBOX_0 &\rightarrow BOX_1\ BOX_2
\end{aligned}
$$

(c) **Interface encoding.**

Figure 2.6: **Interfaces for tree grammars**. Subfigures show manually encoding multiple production right-hand sides, an encoding that uses a Box non-terminal for indirection, and the high-level encoding using interfaces and classes.

```
{"class": "HBox",
 "children": {
   "left": {
     "class": "HBox",
     "children": {
       "left":  {"class": "HBox", "w": 20, "h": 5},
       "right": {"class": "HBox", "w": 15, "h": 5}}},
   "right": {
     {"class": "HBox", "w": 15, "h": 5}}}}
```

Figure 2.7: **Input tree as a graph with labeled nodes and edges**. Specified in the JSON notation.

```
interface BoxI {
  var x : float;
}
class HBoxLeaf : BoxI {
  attributes {
    var y : int;
    input w : ? int;
    input h : int = 10;
  }
}
```

Figure 2.8: **Input tree as graph with labeled nodes and edges**. Specified in the JSON notation.

edge labels specify child bindings. Figure 2.8 uses the JSON format common to dynamic languages for an instance of a tree in `H-AG` . By naming children, such as left and right, we eliminate sensitivity to their order within a code block. With order sensitivity, adding a middle child center would needlessly require refactoring references to the repositioned element right. Likewise, reordering children in the input data does not require refactoring the attribute grammar.

Our second departure from the canonical attribute grammar encoding optimizes the data representation by eliding intermediate interface nodes. The reduction to attribute grammars suggests adding a new non-terminal for each interface, but doing so in the data representation doubles the number of nodes in the concrete tree. Making the interface pattern a language construct with compiler support eliminates associated costs, such as cutting file size for runtime parsing of big data visualizations.

Interfaces for attributes and information hiding.

Our system provides lightweight specification annotations for different types of attributes, and coupled with the interface construct, it supports defining relationships between attributes across different classes.

Each static attribute is annotated with its assignment type and its embedded value type:

```
1 trait Rectangle {
2   attributes { render : int; }
3   actions { render := paintRect(x,y,w,h, "black"); }
4 }
5 class HBox(Rectangle) : BoxI { ... }
```

Figure 2.9: **Trait construct**. Adds shared rendering code to the HBox class.

- **Assignment types.** The assignment type denotes whether the input tree defines the value, such as in input w, or whether the attribute grammar defines it, as in var x. Assignments to an input type are illegal, and multiple assignments to a variable type are also illegal.

  If an input tree fails to provide an input attribute, a runtime error will be thrown. To still provide an interpretation of such trees, input attributes support the annotation "?", which enables pattern matching through functions maybeReady :: > boolean and maybeValue :: > 'a. Alternatively, for the common scenario of using a fixed default value, a default value can instead be defined as in input h : int = 10. If the input tree does not provide the value, the default value will be automatically substituted.

  Canonical attribute grammars can encode input attributes in two ways. First, semantic functions with no parameters encode the lack of dependencies. Second, for finite domains, the set of tree grammar productions can expand to include attribute nodes. The second encoding more faithfully describes our approach because, like our system, it feeds into an automatic tree parser generator. For each tree node, our generated parser scans for the expected set of input attributes.

- **Value types.** The system also supports type annotations used for embeddings. Generated code typically compiles as part of a project in a more static language, such as C++, which require a static type discipline. The annotations can be user-defined, such as OpenGL's *vertex buffer object* VBO, which is not defined within our system.

  Our analyzer ignores the value type annotations such as x : float and y : int while the low-level code generator passes along the decorations float and int. The embedded design simplifies implementation because value type checking is performed by the host language's compiler.

In practice, we use attribute definitions in interfaces for information hiding across classes and lightweight specification of relationships between similar classes. An attribute declared inside of a class is *local* to constraints in the class: only the class's constraints can read or write to the attribute. Conversely, declaring a *var* inside of an interface hints that it is meant to be reused by outside classes, such as part of a tree traversal.

```
1  interface BoxI {
2    var w : int;
3    var h : int;
4    var right : int;
5    var bottom : int;
6  }
7  class HBox : BoxI {
8    children {
9      childs : [ BoxI ]
10   }
11   actions {
12       loop childs {
13         w := fold 0 .. self$ .w + childs$i.w;
14         h := fold 0 .. max(self$ .h, childs$i.h)
15         childs.right := fold x .. childs$ .right + childs$i.w;
16         childs.bottom := fold y .. childs$ .bottom + childs$i.h;
17       }
18   }
19 }
```

Figure 2.10: **Input tree as graph with labeled nodes and edges**. Specified in the JSON notation.

Traits: Reusing Cross-cutting Code

As with many object systems, we support a trait construct for cross-cutting code that should be shared across classes. It statically expands like a macro, and therefore provides no formal expressive power. For example, Figure 2.9 defines how to render a rectangle given several attributes, and then adds that functionality to class HBox. If the language was extended with class VBox, the class definition of VBox could also use trait Rectangle.

Loops

We extend our language with declarative loops over the attributes of multiple nodes. They are an expressive extension over the uniform recurrence relations of [[CITE]].

The loop construct, loop, specifies a block of loop body statements. It acts over a sequence of nodes declared with the same interface, such as childs : [BoxI] in Figure 2.10. The looping order is restricted to forward iteration, though our approach generalizes to other loop orders.

A statement in a loop body will execute for each element of the list. For example, the following statement assigns the attribute w the sum of the children widths: w := fold 0 .. self$ .w + childs$i.w Similar to array index notation, the suffix on righthand-side variable names for loop statements provide a restricted form of relative indexing for loops. In particular:

- $i: the "current" loop step
- $-: the previous loop step
- $$: the last loop step

Use of suffix "$-" in a fold can be thought of as an accumulator in functional programming.

One loop statement can refer to the accumulator of another, which fold statements in most languages do not support. For example, two loop counters can be intertwined:

```
1 loop childs {
2   childs.counter1 := fold 0 .. childs$ .counter2 + 1;
3 }
4 loop childs {
5   childs.counter2 := fold 0 .. childs$ .counter1 + 1;
6 }
```

The programmer does not manually order the statements. For example, our system infers that the imperative code that implements the above declarations is just one imperative loop that fuses them together. The incorrect alternative of implementing the declarations as a different imperative loop for each would lead to unfulfilled data dependencies.

We reduced scheduling loops to scheduling canonical attribute grammars. Our insight is that, for a restricted language of relative indices, we can schedule several unrolled loop steps and generalize the schedule to the rest. Section 3.3 discusses this in more detail.

The declarative nature of the loop construct provides two key benefits. First, coupled with the restricted indexing language, underspecification of the statement order provides freedom for automatic parallelization (Section 3.3). Second, it allows programmers to choose how to structure the program. For example, separating loop statements as above might improve legibility if they are for two different purposes, but as the computation is more intertwined, the programmer has the freedom to choose the following formulation instead:

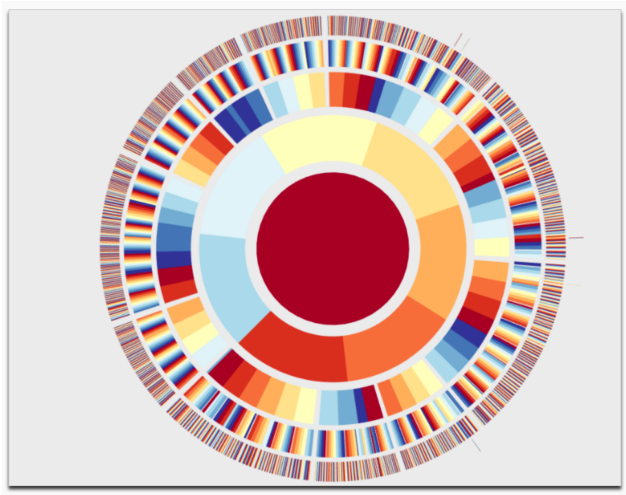
```
1 loop childs {
2   childs.counter2 := fold 0 .. childs$ .counter1 + 1;
3   childs.counter1 := fold 0 .. childs$ .counter2 + 1;
4 }
```

The formulation brought the two statements together and changed their lexical order. Our language guarantees that such a refactoring does not change the semantic meaning of the code.

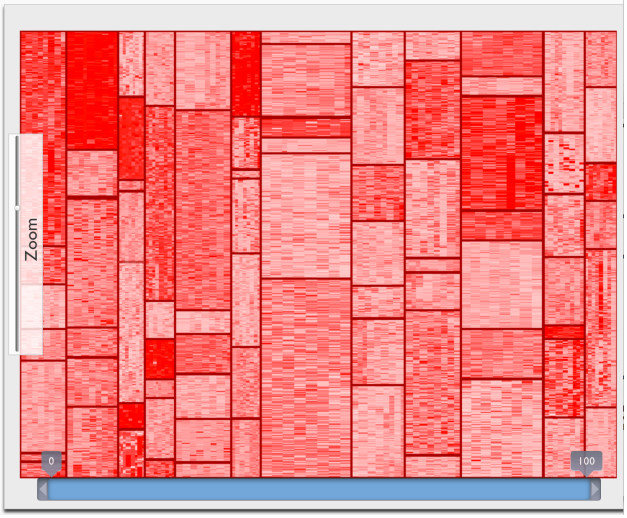
Embedded Domain Specific Language: Functional Rendering

We designed our system for interaction with other tools and languages. A key ability is to invoke externally-defined functions, such as max() of Figure 2.10 for the maximum of two numbers and paintRect() of Figure 2.9 to draw a rectangle to the screen. Attribute grammars are compiled to run in some host system, such as JavaScript or OpenCL, and any function in scope to the generated code may be called.

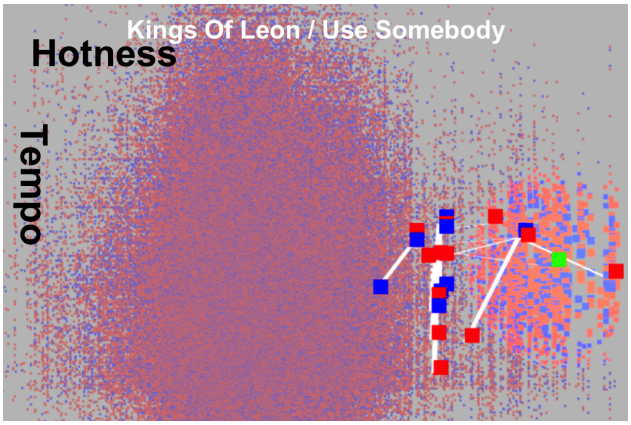
Functions can be safely embedded as long as they provide a *pure* interface. In particular, the returned output should only depend on the inputs. Likewise, functions should be reentrant for use in automatic parallelization In the case of embedding in statically checked languages, the host's static checker is responsible for checking usage.



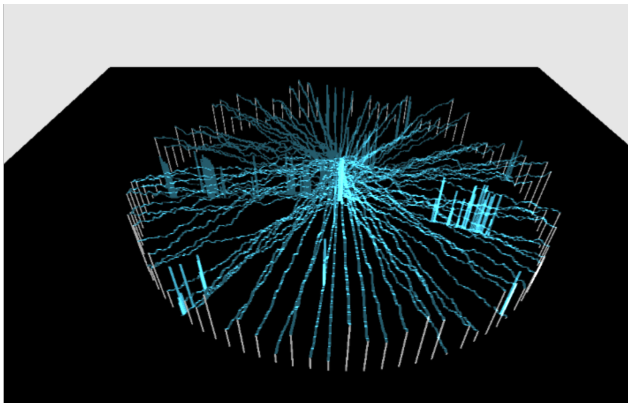
(a) **Sunburst**



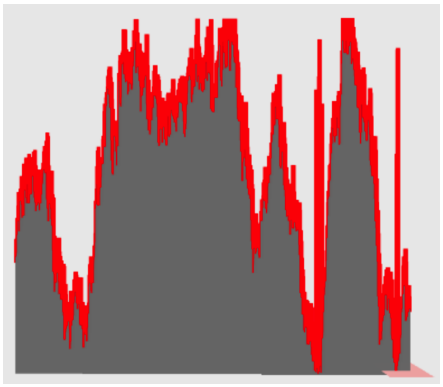
(b) **Treemap**



(c) **Linked Scatter**

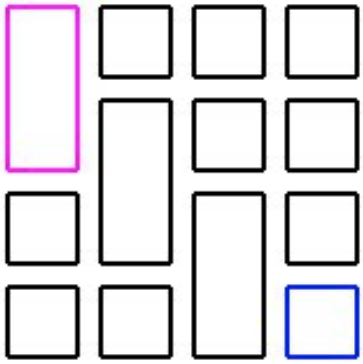


(d) **3D Multiple Time Series**



(e) **Line Graph**

Figure 2.11: **Visualization screenshots.** All except [[CITE]] are interactive or animated. Each one was declaratively specified with our extended form of attribute grammars and automatically parallelized. Labels describe whether GPU or multicore code generation was used.



(a) **HTML Tables** (grid-based)

(b) **CSS** (flow-based)

Figure 2.12: **Document layout screenshots.**

2.4 Evaluation: Mechanized Layout Features

We specified many common layout language features with our extended form of attribute grammars. Most examples were written with few, if any, modifications to the generated code. This experience shows that our restricted form of attribute grammars are a viable formalism for layout specification. The following subsections present highlights from our case studies in specifying layouts with attribute grammars, and the appendix contains the full code.

Rendering

We found several rendering patterns to be important for many visualizations. A library of functional graphics primitives, such as paintRect in Figure 2.9, sufficiently augmented our attribute grammar language in order to achieve them.

- **2D and 3D.** Our base primitives are 3D, and we provide 2D primitives that reduce into them.
- **Color.** Our functional graphics primitives take an RGBA value as input, which enables controlling hue, luminosity, and opacity.
- **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) ;
```

Non-Euclidean: Sunburst Diagram

Visualizations often require non-Euclidean layouts, such as the polar layout for the Sunburst diagram. Instead of propagating and computing over Euclidean values such as x and y coordinates as in `H-AG` , the visualization can use some other.

For example, in a sunburst diagram (Figure 2.11a), a node should be rendered far from the center of the chart if its level is high. In our implementation, each node transitively computes its radius as a function of its parent's. Likewise, the center of visualization propagates from parent to child, with the root node representing the center:

```
1  class Radial : Node {
2    ...
3          loop child {
4                child.parentTotR := parentTotR + r;

6                child.rootCenterX := rootCenterX;
7                child.rootCenterY := rootCenterY;
8          }
9          ... Arc(rootCenterX, rootCenterY, show * (parentTotR + r), ...);
10 }
```

The full example is available in Appendix **??**.

Charting: Line Graphs and Scatter Plots

We specified several types of charts with attribute grammars. For example, Figure **??** depicts an X/Y scatter plot and Figure 2.11e depicts a line graph. We represent every data point as a leaf node in the tree. Tree traversals will compute details such as the X and Y ranges of a data set, which facilitates features such as normalization and centering.

Time series charts led to two additional encoding tricks. First, multiple time series data should often be represented at the same time, such as for a server farm, the output of each server as the days pass. Figure 2.11d depicts one such multiple time series chart. Our approach was to represent each line as an intermediate node:

```
1  class Root : Root I {
2    children {
3       lines : [ LineI ];
4    }
5  }
6  class Line : LineI {
7    children {
8      points: [ PointI ]
9    }
10 }
11 class Point : PointI { }
```

Second, we found the above (Section 2.4) rendering features such as zooming, panning, and 3D representations to be important for visualizing big time series data.

Animation and Interaction: Treemap

We declaratively encoded various animation effects with attribute grammars. For example, the fisheye effect enlarges the size of an element the closer the mouse draws near to it. Our core pattern is to encode time varying values as such the mouse position as input attributes and rerun the layout solver when the inputs change.

Beyond human interaction, we also support reaction to time. For example, for the treemap shown in Figure 2.11b, users may change the data set shown. Instead of immediately showing the new data set, we introduce a tween attribute that an animation increments over time from 0 up to 1. The treemap interpolates the layout position based on the time, which yields a smooth transition for each data point:

```
1  class Point : PointI {
2    attributes {
3      input startW : int;
4      input endW : int;
5      var w : float;
6      var tween : float;
7    }
8    actions {
9      ...
10     w := startW * tween + endW * (1.0f   tween);
11     render := paintRect(x, y, w, h, ...
```

Visualizations like the treemap require recompilation of most of the attributes for such animations, which can become a bottleneck and thus benefits from acceleration by our tool.

Grid-based: Tables

We now examine one of our most difficult case studies: HTML [[CITE]] and CSS table layout [[CITE]]. Tables appear in most rich document layout languages such as CSS and LaTeX, and are an instanced of *grid-based layout*, which is popular for representing layouts such as user interfaces and data tables. In conversations with commercial browser developers, we found that the proposed standards for the layout language features were reverse-engineered from earlier implementations. Furthermore, at the time of writing, two such competing standards were proposed, and with unclear notions of completeness or cases of distinction.

We found that specifying tables involved *non-linear* reasoning about *dynamic DAGs*, which we achieved by using abstract data types and using encoding hints to perform DAG scheduling by reusing our attribute grammar tree scheduler. More dynamic formalisms such as a higher-order attribute grammars [**??**] provide flexible alternatives, but it is not clear how to use them to address the performance criteria of the subsequent chapters.

Several challenges emerged in our analysis of HTML tables:

- **Dynamic data structure.** Layout constraints guide the mapping from a cell node to its location in the table. The computed result of attribute constraints therefore determines the underlying graph structure rather than being provided as part of the input.

- **Computing over a DAG rather than tree.** Each cell of a table has two parent nodes: its row and its column. Static attribute grammars are more typically designed for computations over trees, where each node has at most one parent. Reasoning about dependencies must support this new structure.

- **Non-linear constraints.** Static attribute grammars linearly bound the computation size in terms of the number of attribute instances. A more iterative process is instead used to compute dimensions for CSS's automatic table layout algorithm.

Ultimately, we wrote table-specific code in the specification (see above) and the runtime, but no table-specific code in our scheduler nor code generator. For an example of logic in the specific, the specification constructs the grid data structure by manipulating functional lists rather than just numbers. Likewise, to ensure a column's computations over its cells are scheduled after the grid is constructed, we included this dependency in the specification.

Our runtime edits were to use a breadth-first traversal for traversing a table and, to lookup the children of a column, search table rows for cells with the corresponding column number attribute. We did not have to add table-specific code into the synthesizer (the offline scheduling analysis) nor the code generator.

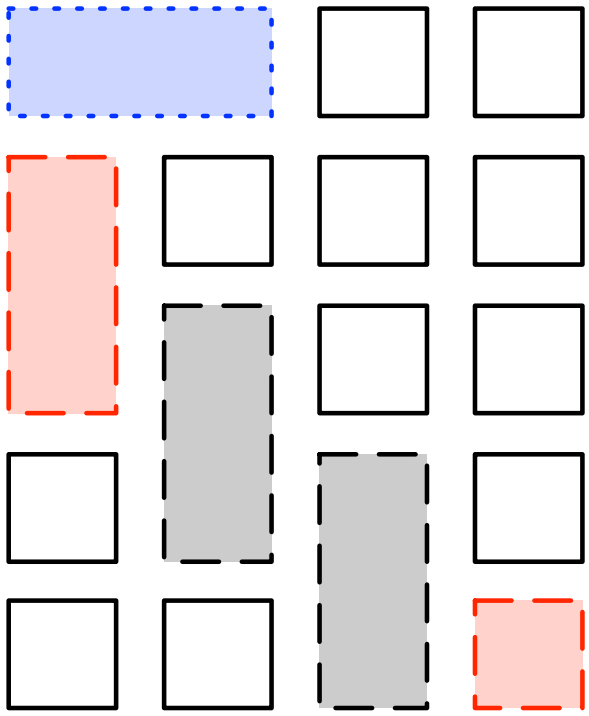
We address each problem in turn.

Dynamic data structure.

Figure 2.13 illustrates why the mapping from table cells to table column is dynamically computed. The placement of a cell is complicated by preceding cells that span multiple rows ("rowspan=n") and columns ("colspan=n"). Ultimately, the cell must be placed in the first column such that an earlier cell in a top-down, left-to-right ordering does not overlap it. The figure shows two important cases. First, the second cell of the first row is placed in the third column because its left sibling spans two rows: a cell's column is a function of the rowSpan attributes of its siblings to the left. The second case is shown for the bottom right cell. Even though it is the third cell of its row in the parse tree, it is not placed in the third column. The reason is that the red dashed rectangular cell in the second row transitively impacts the placement of the cells after it. The colSpan attributes of cells in rows above a cell further determine its column.

Our specification computes the column assignment as a loop over the rows. For each row, it computes what columns its cells are placed in as a function of the list of columns that are still occupied by preceding cells. The next row is given the columns that are occupied after adding cells on the current row, etc. Our specification of this behavior is interesting in that it is just calls to functional list manipulation methods written in our host language:

```
1  class TableBox
2    ...
3    loop rows {
4      rows.colAssignment :=
5        fold
6          emptyColumnList(colCount)
```



(a) **Table Rendering**

```
<table>
  <row>
    <cell rowSpan="2"></cell>
    <cell></cell>
    <cell></cell>
  </row>
  <row>
    <cell colSpan="2"></cell>
    <cell></cell>
    <cell></cell>
  </row>
  <row>
    <cell colSpan="2"></cell>
    <cell></cell>
    <cell></cell>
  </row>
  <row>
    <cell></cell>
    <cell colSpan="2"></cell>
    <cell></cell>
  </row>
  <row>
    <cell></cell>
    <cell></cell>
    <cell></cell>
  </row>
```

(b) **Textual encoding of input tree.**

Figure 2.13: **Document layout screenshots.**

```
Schedule {
  Col.childs[i].relX < Col.cellsready
  Col.childs[i].absX < Col.cellsready
```

(a) **Surface Syntax** (Proposed)

```
schedule {
  asserta(assignment(col, self, childs_relx_step, self, cellsready)),
  asserta(assignment(col, self, childs_absx_step, self, cellsready))
```

(b) **Low-level constraint**

Figure 2.14: **Specifying dynamic dependencies.**

```
        ..
columnsAppendRow(
  rows$ .colAssignment,
  rows$i.cells,
  rows$i.rowNum);
```

The columnsAppendRow function computes the column position during placement, so subsequent reads can look it up through another list manipulation function.

A column computes the x coordinates for each cell, but column cells are not known before

the last columnsAppendRow() call. To ensure a column computes over its cells after the mapping occurs, we explicitly declare the dynamic data dependency in the specification. First, the grid is stored in an attribute, so we simply propagate the grid to all the table nodes as an attribute (cellsready). We then state the implicit data dependency (Figure 2.14. The scheduler now knows to run column computations over cells only after the cellsready is computed. Currently, we directly specify the constraints by enabling low-level schedule constraints (Figure 2.14b and Section [[[???]]]), which might be directly generated from surface syntax (Figure 2.14a).

Computing over a DAG

Computing over a table means computing over a DAG, not a tree: a cell has both a row and a column as its parents. This impacted both our runtime and our specification strategy. Demonstrating the flexibility of attribute grammars, we did not have to modify the scheduler nor the code generator. Instead, we modified the runtime and the specification.

We modified the runtime to generalize an important invariant from tree traversals to DAG traversals. In a top down traversal of a tree, a node's parent is visited before the node itself. A valid implementation for trees is depth first. However, consider a depth first traversal of a table's parse tree:

```
1 <table>
2   <row>
3     <cell></cell>
4   </row>
5   <column></column
6 </table>
```

The depth-first traversal would visit the table, the row, the cell, and then the column. The cell is visited before its parent column!

Our modification was simple: we edited the runtime to visit the nodes of a table with a breadth first traversal. We kept the overall document traversal as depth first for performance reasons. Declarative schedule constraints would also support picking a breadth-first traversal (Section **??**).

We also modified the specification to pass our attribute grammar static checker. The changes enables relaxing the scheduler's obligation to guarantee that visiting a cell's parent row and column would set all the attributes needed by the cell (unambiguous) and without conflicting with each other. For example, a column defines the relX attribute of its child cell, and a row, its relY. By default, our checker would rightfully reject such a specification because, if a cell has only one parent, only one of those attributes would be set.

We extended the specification language for instructing the scheduler that external code defines certain attributes:

```
1 class Col : ColI {
2   phantom {
3     childs.relY;
4     childs.absY;
5 ...
6 class Row : RowI {
7   phantom {
```

```
 8       childs.relX;
 9       childs.absX;
10   ...
```

The scheduler now assumes that the external code provides definitions for a column's childs.relY and childs.absY and a row's childs.relX and childs.absX. Unimportant to the synthesizer, the definitions just happen to come from elsewhere in the same specification, such as class Row defining the phantom attributes not set by Column.

Non-linear constraints

The table specification defines a dynamically determined number of loops over a table's column to determine column widths. Such dynamism is beyond the pure static attribute grammar formalism, but our foreign function interface sufficed while still allowing overall specification and scheduling through attribute grammars.

Flow-based: CSS Box Model

Document layout languages generally feature a *flow-based* layout model where the position of one element is largely a function of the previous one. For example, line wrapping places one word after another in a paragraph, and a column will stack one paragraph after another. However, ambiguity quickly arises once constraints are added to such systems. We found that, before being able to address our interest in parallelizing the CSS language, that creating a functional specification of it was already a challenge to itself. This section focuses on the ability to express the CSS specification, and defers discussion of functional correctness (Chapter ??) and safe parallelization (Chapter ??).

Challenging specification, the CSS standard provides only a few explicit formulas such as min(max(intrinsMinWidth,maxWidth),intrinsPrefWidth) for the shrink-to-fit calculation. It generally does not fully define the intrinsic dimensions to plug into the formula. We incorporated what we found, and for the rest, spent significant time reengineering the semantics by examining the standard and experimenting with existing browsers. While it is unclear how to evaluate faithfulness, we encoded enough features to render a resemblance of the Wikipedia main page (Figure **??**) and a popular blog.

Our attribute grammar describes the layout solving features of the informally written CSS 2.1 standard. It also includes automatic table layout, which was only more completely defined in later CSS standards. It does not include preprocessing steps, such as the CSS cascade that annotates the HTML tree with attributes, nor anonymous content generation, which normalizes the annotated tree to guarantee that spans of sibling nodes are homogeneous. The former is largely a combination of a simple extension to regular expressions and prioritization constraints. We found we could include parts of the cascade in our approach, such as handling units, and thus do. Normalization is a bottom-up tree rewriting pass, and an implementation optimization avoids performing it before layout and instead makes it an on-demand part of

layout solving. We primarily focus in the core box model: normal flow (blocks and inlines), out of flow (relative and absolute positioning, floats), and borders, padding, and margins.

Our specification largely follows the style of the above grammars. Part of the intuition for the feasibility of specifying CSS in this way is that CSS was designed with restrictions that avoid requiring slow evaluation with techniques such as iterative constraint solving. In our encoding, each CSS display type is represented by one or more classes in our system. CSS's normalization algorithm largely leads to our set of interfaces, such as grouping the inline and inline block display types under interface inline. We make heavy use of traits and interfaces, which compromise 23% and 32% of the code, respectively. The automatic table layout algorithm was an extension of the above techniques. Finally, similar to the issue with table cells having two parents, a row and a column, out of flow elements also required encodings to support DAG behavior.

Several differences distinguish our experience with specifying CSS layout from the other case studies. Many features were difficult to specify because of many cases or cross-cutting in their semantics. Discussed in Chapter [**??**], we rely upon automatic checking to assist development, and discussed in Chapter [**??**], we specify schedule sketches to improve compiler speed and more quickly experiment with parallelization schemes. To further simplify development, we wrote several increasingly large specifications and manually integrated them.

One particularly challenging feature to disentangle relates to ambiguity. CSS solves seemingly inconsistent input constraints instead of returning an error. For example, if `H-AG` was extended to support input heights on intermediate nodes, the following conflict would require a graceful interpretation rather than refusing to render:

```
1 <hbox height="5">
2   <hbox height="500"></hbox>
3 </hbox>
```

By the original attribute grammar, the outer <hbox> should be the size of the biggest child, which would be 500. However, that conflicts with the input constraint of the outer box only being 5 tall. Our CSS grammar inspects for the presence of input attributes and prioritizes them. The analogous resolution for the `H-AG` example is the following:

```
1 loop children {
2   h :=
3     fold (maybeReady(height) ? maybeValue(height) : 0)
4     ..
5     maybeReady(height) ? maybeValue(height) : max($.h, child.h)
6 }
```

The grammar uses "5" and "500" because they were explicitly specified instead of solving for them.

We found other features to be difficult because they purposefully stray from the direct mathematical interpretation. For example, CSS supports input constraints where a node's width is defined as a proportion of its parent's. If we naïvely extended `H-AG` with such a feature, evaluation of the following layout would lead to a degenerate solution:

```
1 <hbox>
2   <hbox width="50%">
```

```
3         <hbox w="20"></hbox>
4     </hbox>
5   </hbox>
```

The root node shrinks to fit the middle node, but the middle node must be 50% of the parent. Direct interpretation leads to a solution of 0 for both widths, but CSS instead leaves the result up to the layout engine implementation. The first reason is that the result looks unappealing: the containers of the leaf node do not appear. The second reason is that, while iterative solvers may avoid some such situations, but at the expense of performance. Implementations instead use non-iterative heuristics, but as seen with tables, implementors struggle to understand them.

In summary, our attribute grammar formalism was sufficiently flexible to specify a non-trivial subset of the widely used CSS language. We encountered several key difficulties, and discuss those relating to correctness, safe and effective parallelization, and compiler speed in Chapters **??** and **??**.

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 Static Scheduling Language for Parallel Tree Traversals

We now introduce a static scheduling language for parallel computations over a tree such as those seen in layout. A compiler takes an attribute grammar (the functional specification) and its schedule and outputs an implement, i.e., a layout engine. For now, we ignore the source of the schedule. Beyond presenting the language and compilation strategy, we show how to verify functional and behavioral (parallel) correctness, introduce the first parallel schedule for a large subset of CSS, and show how, on GPU tree computations, memory may be dynamically allocated quite efficiently.

The goal of our language for traversals over trees is to benefit from high-performance computing techniques commonly used for stencils over grids. Common for physical modeling, a stencil computation is one where a kernel only reads and writes to a limited set of nodes, there are many kernels, and the data dependencies between kernels lead to traversal patterns such as a wavefront. Many variants of stencils exist. The enthusiasm for them stems from programmers or stencil compilers being able to exploit knowledge of their structure to effectively optimize for use of cache lines, memory, parallel processors, and other resources. Similar techniques are known for trees, and the next chapter introduces additional ones.

Our first challenge in this chapter is to restrict the scheduling language enough to facilitate optimization while still providing the flexibility for expressing document layout and data visualization workloads. Finding parallelism in CSS is already novel, so being further able to optimize it with techniques associated with small formulas for physical models is surprising. We found that, while layout computations have too many data dependencies to be solved with one simple tree traversal, sequences of 3–5 traversals often suffice for data visualizations and 9 for CSS. Our scheduling language therefore consists of traversal patterns, such as parallel top down (*preorder*) traversal of the tree, and ways of combining them, such as in a sequence. One especially representative case study of matching restrictions with flexibility was in our study of visualizations: dynamic memory allocation on a GPU is generally a bottleneck, but we used a sequence of tree traversals for parallel allocation of render buffers for each node.

Another artifact of layout specifications being magnitudes bigger than stencil formulas

is that the correctness concerns change. For stencil computations, the verification challenge lies more in correctly optimizing the implementation of a traversal schedule. Layout computations encounter a challenge before that point: the size of the functional specification and the ensuing tangle of data dependencies require ensuring that the parallel schedule itself is safe to implement. We use a variant of existing static dependency analyses of attribute grammars to verify that the schedule is race-free.

The dependencies that complicate reasoning about correctness of parallel code actually also complicate sequential code. Our use of static analysis for the attribute grammars leads to an important result for layout languages: we show how to statically verify three important properties about them.

- **Totality** The layout language defines a solution for every syntactically well-formed input tree; it is unambiguous.
- **Determinism** As discussed in the previous paragraph, parallelization is safe.
- **Linearity (Single Assignment)** Every attribute is assigned to exactly once. Layout languages often perform *reflow* to iteratively solve constraints or incremental computation, so this property bounds the need for it.

The first property demonstrates the ability to reason about *functional correctness* and the last two about *behavioral correctness*. Put together, we verify that a language is unambiguous, supports parallelization, and with bounded asymptotic complexity.

3.1 Language of Static Schedules

This section focuses on defining our full language of traversal schedules. It is the input for our code generators. Programmers either do not specify the schedule in practice due to our automation support, or use the *sketching* extension (Section 4.1) for more succinct partial specification.

Statically scheduled evaluation departs from the dynamic evaluation strategy of Chapter 2. Static scheduling solves the performance problem of dynamic evaluation repeatedly manipulating the data dependencies of every attribute at runtime. For example, what would be a direct sequence of arithmetic statements in a static language becomes an interleaving of graph manipulations and arithmetic with the dynamic evaluator. The runtime scheduling overhead for evaluating all the statement is (at least) linear in the size of the data dependency graph.

Instead, a schedule statically specifies most of the scheduling decisions. It specifies a sequence of tree traversals and the order of statements to use within each traversal. During a traversal at runtime, the order of nodes to traverse is based on the traversal pattern, such as top-down, rather than by inspecting data dependencies. Likewise, the statements to execute for a node are looked up based on the node's type rather than by data dependencies. Our approach is a more compositional variant of others. Our flexibility requirements led to

focusing on the ability to compose different types of traversals, such as by sequencing and nesting.

Sequential Schedules

We start by examining how to specify a safe static schedule for `H-AG` that respects any possible dynamic dependencies (Figure 2.3a).

Figure 3.1 shows a sequential implementation of `H-AG` decomposed into several pieces. The layout engine solves an input tree over a sequence of two traversals (Figure 3.1a). The first traverses the tree in postorder, meaning from the leaves up to the root ("bottom-up") and the second performs are preorder traversal, meaning from the root down to the leaves ("top-down"). Figure 3.1b provides a sample implementation of generic traversal code. During a during traversal, each node is *visited* exactly once in order to compute the attributes whose dependencies have been satisfied. Figure 3.1c shows that the first pass computes widths and heights and the second pass computes the x and y positions.

The example follows a static schedule rather than manipulating a dynamic data dependency graph. The sequence of traversal invocations and the code used for the different cases for each traversal's visitor determine the schedule. Each traversal now only performs dynamic scheduling in the sense of maintaining a stack for recurring down the tree, which is a cost proportional to the number of nodes rather than the size of the dynamic dependency graph between attributes. In practice (Chapter 5), our compiler and runtime optimizations even eliminate the example's implicit use of a call stack.

The decompose the schedule into several types of policy fragments. First, the schedules involves a *sequence* of *two different types* of traversals:

```
1        postorder(visit1, start); preorder(visit2, start)
```

Just one bottom-up traversal cannot compute all the attributes, such as all the x and y attributes that flow downwards (Figure 2.3a), so the schedule may require multiple types of traversals and in a careful order. Furthermore, within a traversal, the schedule specifies different orders of statements for different types of nodes. Consider the following fragment:

```
1        HBOX$_1$.x = HBOX$_0$.x;
2        HBOX$_2$.x = HBOX$_0$.x + HBOX$_1$.w;
```

The schedule specifies that $\text{HBOX}_2.\text{x}$ can (and should) be immediately evaluated after $\text{HBOX}_1.\text{x}$ without fear of unsatisfied data dependencies for any of its right-hand side terms. In summary, we see three parts to a schedule: the staging of traversals, the node visit order for every individual traversal, and the statement order for different types of nodes within a specific traversal.

We abstracted the three aspects of a schedule into a scheduling language (Figure 3.3). For example, the schedule for the above computation would be appear as:

```
1  postorder
2    HBOX0 → HBOX1 HBOX2 { HBOX0.w HBOX0.h }
3    HBOX → ϵ { HBOX.w HBOX.h }
4  ;
```

```
1  postorder(visit1, start);
2  preorder(visit2, start);
```

(a) **Sequential sequence of traversals**

```
1  void preorder(void (*visit)(Prod &), Prod &p) {
2    visit(p);
3    for (Prod rhs in p)
4      preorder(visit, rhs);
5  }
6  void postorder(void (*visit)(Prod &), Prod &p) {
7    for (Prod rhs in p)
8      postorder(visit, rhs);
9    visit(p);
10 }
11 void recursive(void (*visit)(Prod &, int), Prod &p) {
12     int step = 0;
13     visit(p, step++);
14     for (Prod rhs in p) {
15       recursive(visit, rhs);
16       visit(p, step++); //repeat visit to p
17     }
18 }
```

(b) **Three sequential traversal patterns**

```
1  void visit1 (Prod &p) {
2    switch (p.type) {
3      case S → HBOX: break;
4      case HBOX → ϵ:
5        HBOX.w = input(); HBOX.h = input(); break;
6      case HBOX → HBOX1 HBOX2:
7        HBOX0.w = HBOX1.w + HBOX2.w;
8        HBOX0.h = MAX(HBOX1.h, HBOX2.h);
9        break;
10   }
11 }
12 void visit2 (Prod &p) {
13   switch (p.type) {
14     case S → HBOX:
15       HBOX.x = input(); HBOX.y = input(); break;
16     case HBOX → ϵ: break;
17     case HBOX → HBOX1 HBOX2:
18       HBOX1.x = HBOX0.x;
19       HBOX2.x = HBOX0.x + HBOX1.w;
20       HBOX1.y = HBOX0.y;
21       HBOX2.y = HBOX0.y;
22       break;
23   }
24 }
```

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

Figure 3.1: **Sequentially scheduled and compiled layout engine for `H-AG`.**

```
5  preorder
6    S → HBOX { HBOX.x HBOX.y }
7    HBOX0 → HBOX1 HBOX2
8      { HBOX1.x HBOX2.x HBOX1.y HBOX2.y }
```

It specifies a sequence (";") of two traversals of node visit order postorder and preorder. For each type of node visited within a traversal, the schedule specifies the sequential sequence of attributes to evaluate. We note that, due to the desugaring of our class system in Section 2.3, the dispatches in the above examples are based on grammar productions in the desugared representation. In terms of the fronted language, the dispatches are based on node class.

Generally, a single attribute grammar may be scheduled in many ways. For example, the width and height computations share no dependencies, so the first postorder traversal might be partitioned into two postorder traversals:

```
1  postorder
2    HBOX0 → HBOX1 HBOX2 { HBOX0.w }
3    HBOX → ϵ { HBOX.w }
4  ;
5  postorder
6    HBOX0 → HBOX1 HBOX2 { HBOX0.h }
7    HBOX → ϵ { HBOX.h }
8  ;
9  preorder
10   S → HBOX { HBOX.x HBOX.y }
11   HBOX0 → HBOX1 HBOX2
12     { HBOX1.x HBOX2.x HBOX1.y HBOX2.y }
```

Rescheduling in this way may improve performance on small devices with little memory because the schedule cuts the working set size in half for each traversal. Note, however, that the schedule only optimizes execution; it does not change the result of evaluation.

Sequential execution supports a traversal type that can compute more than postorder or preorder, which we call a recursive traversal (Figure 3.1b). We use a recursive traversal, for example, for line breaking in our document layout case study. Consider inserting line breaks into the following stylized paragraph of XML strings (Figure 3.2):

```
lorom <italic><huge>ipsum dolor</huge></italic> sit
```

Due to `<huge>`, the paragraph may need a line break between "ipsum" and "dolor." Identifying the line break position involves visiting the subtree `<italic>...</italic>`; the resulting line break position is a data dependency influencing line breaks in the remainder of the text. The sequence of arrows in the big circle of Figure 3.2 show a trace of performing a recursive traversal over the paragraph. The traversal visits a node n, then visits n's first child, revisits n, and repeats this process for the remaining children before returning to the parent.

The relationship between recursive traversals and postorder and preorder merits examination. First, a sequence of a preorder traversal followed by a postorder traversal may be merged into one recursive traversals. Traversing a tree induces overhead costs, so such fusion may be beneficial. The reverse relationship is not true, however. As happens with the case of line breaking, long-running sequential dependencies may prevent splitting a recursive traversal into a preorder and postorder traversal. These dependencies arise because the the same node

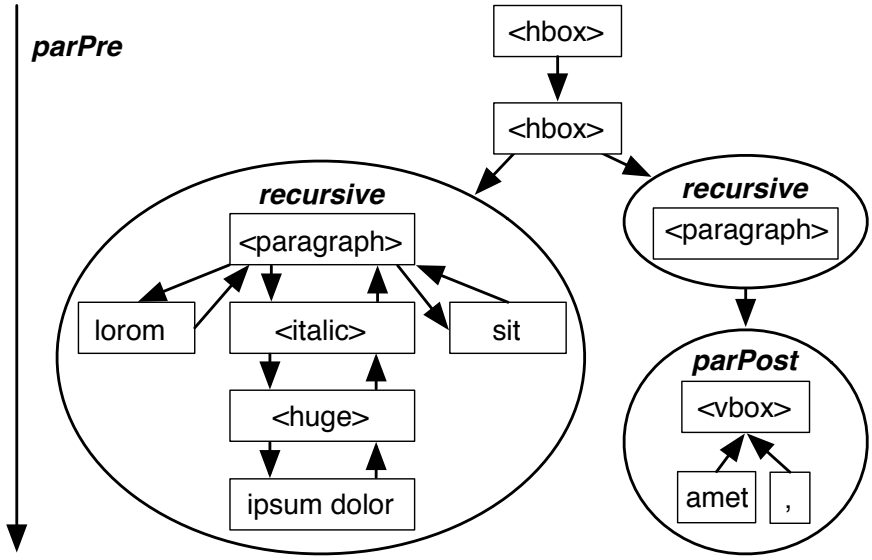


Figure 3.2: **Nested traversal for line breaking**. The two paragraphs are traversed in parallel as part of a preorder traversal. A sequential recursive traversal places the words within a paragraph. Circles denote nested regions and arrows show data dependencies between nodes and/or regions.

is visited multiple times in a traversal: once before a child subtree is traversed and again after. The result of computing over one subtree may therefore be used to compute another, which supports long-running sequential dependencies.

Parallel Schedules: Same Traversal

A schedule exposes structured parallelism both within a traversal and across them.

For an example of parallelism within a traversal, the first postorder traversal for `H-AG` features latent parallelism. The widths and heights for one subtree can be computed independently of the widths and heights of another distinct subtree. Figure 3.4a shows an example where different (logical) threads may compute on the leaf nodes and implicit barriers force a join at every intermediate node. Likewise, the second traversal (Figure 3.4b) may be changed to a parallel preorder traversal where ever intermediate node acts as a logical fork. Figure 3.3b depicts naïve parallel implementations using Cilk's [**??**] spawn and join primitives. We formulate the schedule by changing the specification from postorder and preorder to parPost and parPre (Figure 3.3a).

Our *nested* traversal feature supports exploiting parallelism within a traversal even if some nodes require sequential evaluation. With it, the tree is partitioned into an outer region and disjoint inner regions. The outer and inner regions are evaluated with different traversals, and both may exploit parallelism. We can think of the inner regions as macro-

```
1  parPost
2    HBOX0 → HBOX1 HBOX2 { HBOX0.w HBOX0.h }
3    HBOX → ϵ { HBOX.w HBOX.h }
4  ;
5  parPre
6    S → HBOX { HBOX.x HBOX.y }
7    HBOX0 → HBOX1 HBOX2
8      { HBOX1.x HBOX2.x HBOX1.y HBOX2.y }
```

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

```
1  void parPre(void (*visit)(Prod &), Prod &p) {
2    visit(p);
3    for (Prod rhs in p)
4      spawn parPre(visit, rhs);
5    join;
6  }
7  void parPost(void (*visit)(Prod &), Prod &p) {
8    for (Prod rhs in p)
9      spawn parPost(visit, rhs);
10   join;
11   visit(p);
12 }
```

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

```
1  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>**)*}?

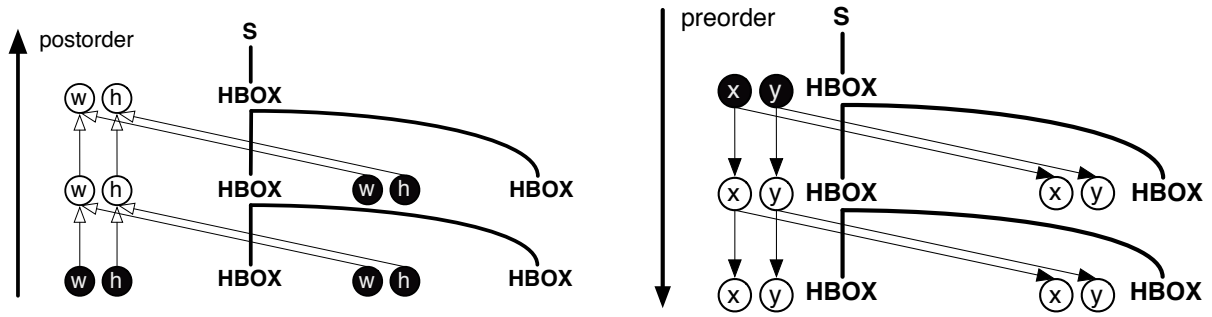
<TravAtomic> → **preorder** | **postorder** | **parPre** | **parPost** | **recursive**

<Visit> → *<Prod>* { *<Step>** }

<Step> → *attrib* | **recur** *v*

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

Figure 3.3: **Scheduled and compiled layout engine for `H-AG`.**



(a) First traversal: parallel postorder

(b) Second traversal: parallel preorder.

Figure 3.4: **Parallel Traversal**. Shown for constraint tree in Figure ZZZ (a). Circles denote attributes, with black circles denoting attributes with resolved dependencies such as input()s. Thin lines show data dependencies and thick lines show production derivations. First diagram shows dependencies followed by first traversal, and second for the following traversal.

nodes that are evaluated in full (with their particular traversal type) when the outer traversal encounters them.

To motivate the need for nested traversals, we revisit line breaking. Even though line breaking of a single paragraph is sequential, distinct paragraphs of text can be handled in parallel. To avoid locally sequential computations from forcing the entire tree traversal to be sequential, we allow the outer region to be parallel, while each paragraph forms an inner region that is handled with the sequential recursive traversal. Figure 3.2 shows how parallel evaluation may be used to compute across different **recursive** paragraphs. Likewise, it shows a hypothetical VBox subtree that uses parallel postorder evaluation for traversing its subtree as soon as the outer parallel preorder traversal reaches it.

To partition a tree into regions, the schedule maps each grammar production (and thus each node of the tree) to a traversal type in the synthesized schedule. A subtree composed from nodes of the same traversal types form an inner region. For example, a nested traversal of paragraphs with sequential traversals of nested text subtrees is described as follows:

```
1  parPre
2    P → W { W.relativeX }
3    { recursive ↦
4        W0 → W1 W2 {
5          W1.relativeX recur W1
6          W2.relativeX recur W2 } }
```

Parallel Schedules: Across Travesals

We may exploit parallelism across traversals as well. For example, just as we created a different but functionally equivalent sequential schedule for `H-AG` , we can do the same

manipulation to yield a new parallel schedule:

```
1  (    parPost
2         HBOX0 → HBOX1 HBOX2 { HBOX0.w }
3         HBOX → ϵ { HBOX.w }
4       ||
5         parPost
6         HBOX0 → HBOX1 HBOX2 { HBOX0.h }
7         HBOX → ϵ { HBOX.h })
8  ; parPre ... /* same as before */
```

The "||" construct specifies that one traversal may be run concurrently with another. Neither traversal traversal depends on attributes written by the other, so the parallelizaton is safe. Even if we cannot exploit parallelism within a traversal, using "||" enables us to exploit parallel across them.

Compilation

Compilation only requires an attribute grammar and the schedule. The traversal staging postorder _ ; preorder _ directly translates to the executable fragment in Figure 3.1a. Likewise, the mapping from traversal productions to statement sequences, such as HBOX → ϵ { HBOX.w HBOX.h }, directly translate to the visit functions of Figure 3.1c. The translation matches an attribute in the schedule with the lefthand side attribute of an equation in the attribute grammar and outputs the full assignment statement in its place.

Our code generation pipeline is further complicated but conceptually similar. The schedule is combined with the attribute grammar to form an intermediate representation, and different code generators target different backends such as JavaScript, OpenCL, and C++. Furthermore, some of the reductions of Section **??** require augmenting or rewriting the intermediate representation, such as reinserting loops that were unrolled during scheduling (Section **??**). Our end-to-end compiler design is slightly different due to the synthesis algorithm (Chapter 4) and schedule autotuning (Section **??**), but code generation for a schedule follows a more conventional process.

3.2 Automatically Staging Memory Allocation for SIMD Rendering

Problem

The static language of traversals is restricted, but we find that it can express important cases of typically more dynamic constructs. Prominent in our case studies, dynamic memory allocation provides significant flexibility for a language, but it is unclear how to perform it on a GPU without significant performance penalties. Our insight is that the memory allocation may be staged with parallel traversals by using a variant of prefix sum node labeling.

```
1 float *drawCircle (float x, float y, float radius) {
2   float *buffer = malloc( (2 * sizeof(float) ) * round(radius))
3   for (int i = 0; i < round(radius); i++) {
4     buffer[2 * i] = x + cos(i * PI/radius);
5     buffer[2 * i + i] = y + sin(i * PI/radius);
6   }
7   return buffer;
8 }
```

(a) **Naıve drawing primitive .**

```
1 int allocCircle (float x, float y, float radius) {
2   return round(radius);
3 }
```

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

```
1 int fillCircle(float x, float y, float radius, float *buffer) {
2   for (int i = 0; i < round(radius); i++) {
3     buffer[2 * i] = x + cos(i * PI/radius);
4     buffer[2 * i + i] = y + sin(i * PI/radius);
5   }
6   return 0;
7 }
```

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

Figure 3.5: **Partitioning of a library function that uses dynamic memory allocation into parallelizable stages.**

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 created two complementary automation techniques. First, we use our synthesizer to automatically schedule parallel memory allocation. Second, we syntactically hide the use of our optimization 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 3.5a. 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 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)

```
1  CBOX → BOX1 BOX2
2  {
3    ...
4    CBOX.render =
5      drawCircle(CBOX.x, CBOX.y, CBOX.radius)
6        + drawCircle(CBOX.x + 10, CBOX.y + 10, CBOX.radius * 0.5);
7  }
```

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

```
1  CBOX → BOX1 BOX2
2  {
3    ...
4    CBOX.sizeSelf =
5      allocCircle(CBOX.x, CBOX.y, CBOX.radius)
6        + allocCircle(CBOX.x + 10, CBOX.y + 10, CBOX.radius * 0.5);
7    CBOX.size = CBOX.sizeSelf +BOX1.size + BOX2.size;
8    BOX1.buffer = CBOX.buffer + CBOX.sizeSelf;
9    BOX2.buffer = BOX1.buffer + BOX1.size;
10   CBOX.render =
11     fillCircle(CBOX.x, CBOX.y, CBOX.radius, CBOX.buffer)
12       + fillCircle(CBOX.x + 10, CBOX.y + 10, CBOX.radius * 0.5,
13             CBOX.buffer + allocCircle(CBOX.x, CBOX.y, CBOX.radius));
14 }
```

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

```
1  CBOX → BOX1 BOX2
2  {
3    ...
4    CBOX.render =
5        @Circle(CBOX.x, CBOX.y, CBOX.radius)
6        + @Circle(CBOX.x + 10, CBOX.y + 10, CBOX.radius * 0.5);
7  }
```

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

Figure 3.6: **Use of dynamic memory allocation in a grammar for rendering two circles.**

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. Figure 3.7 shows the dynamic data dependencies and two parallel tree traversals for an instance of staged parallel memory allocation.

Library functions that requires dynamic memory allocation are manually rewritten into allocation request (Figure 3.5b) and memory consumption fragments (Figure **??**). The trans-

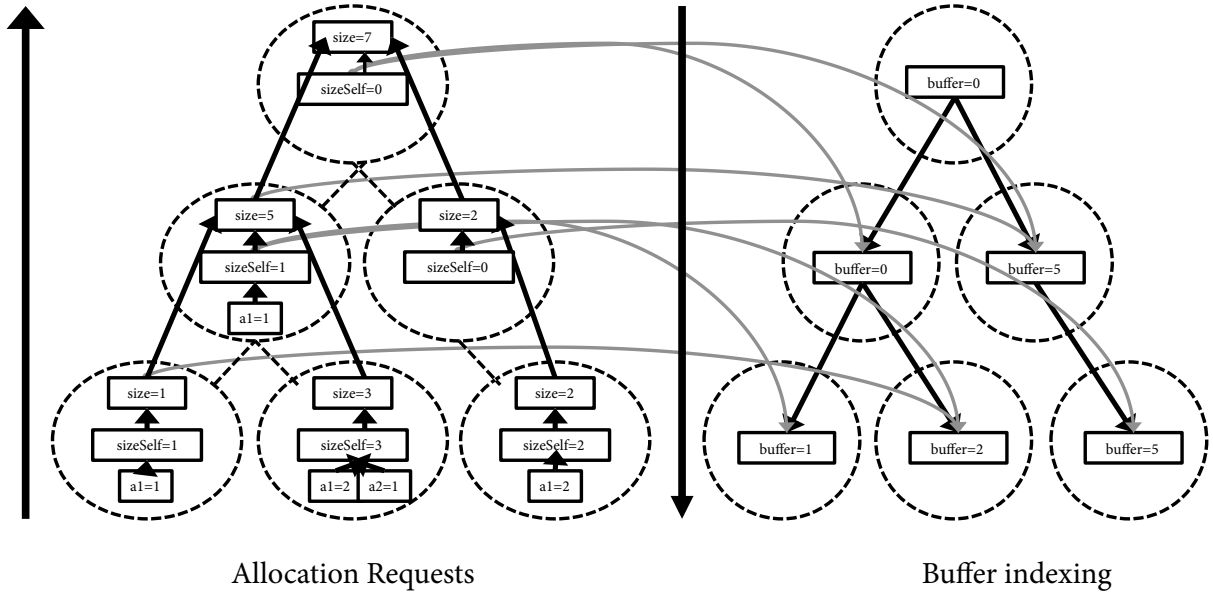


Figure 3.7: **Staged parallel memory allocation as two tree traversals.** First pass is parallel bottom-up traversal computing the sum of allocation requests and the second pass is a parallel top-down traversal computing buffer indices. Lines with arrows indicate dynamic data dependencies.

formation 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 3.6a) is split into calls for allocation requests, buffer pointer manipulation, and buffer usage (Figure 3.6b). 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 3.6c and Figure 3.6a. Our macros (implemented in OMetaJS [[CITE]]) automatically expand into the form seen in Figure 3.6b.

Our use case only required one allocation stage, but multiple may be needed. For example, a final logging stage might be added that should run after all other computations, including rendering. However, the '@' calls described above expand to contribute to one attribute (size): no allocation is made until all of the sizes are known, which prevents making an allocation after using dynamic memory. To support multiple allocation stages, the '@' macro could be expanded to include logical group names: @[render]Circle (...) would contribute to sizeRender, @[log]error (...) to sizeLog, and @[render,log]Strange (...) to both sizeRender and sizeLog. Parallel traversals would be created for each logical name, and the synthesizer would be responsible for determining if the traversals can be merged in the final schedule and implementation.

3.3 Statically Scheduling Loops

Many of the difficulties in computer science stem from handling loops. Our question is how to statically schedule uses of the declarative construct of Section 2.3. The construct extends the language of statements to include non-nested loops, and an attribute computed in one step of one loop may depend on that of another. To avoid implementation complexity, we want to schedule loops through a reduction to a language without loops. Our insight is that we can finitely unroll any loop a fixed number of times in such a way that its schedule generalizes to a loop over an arbitrary number of items at runtime.

Our problem is distinct from that of classical attribute grammar languages for two reasons. First, modern formalisms focusing on expression generally rely upon dynamic scheduling. Second, for the formalisms that provide static scheduling, loops would be over the tree rather than as part of the statement language. For example, a list of values would be encoded as a chain:

<BinaryNode> $\rightarrow$ *<ValueList> <BinaryNode> <BinaryNode>* $\mid \epsilon$

<ValueList> $\rightarrow$ *number <ValueList>* $\mid \epsilon$

The position of a number in a ValueList chain corresponds to the tree level, and so a loop over them will occur as multiple steps of a global tree traversal. However, such loops are generally localized and one instance should be computable as part of the same step. Local loops simplify parallelization and can reduce the number of required tree traversals. Finally, local loops increase the expressivity of a traversal by eliminating what would otherwise represent a non-local dependency that could challenge scheduling as a structured tree traversal.

In terms of the above encoding, our support of loops corresponds to extending ordered attribute grammars with a Kleene star. We could use it to modify the above program to keep a list of values local to a production:

<BinaryNode> $\rightarrow$ *number** *<BinaryNode>* *<BinaryNode>* | ϵ

The language of constraints are recurrence relations [**??**]. Attribute dependencies may lead to subtle interactions with traversal patterns, however. For example, in a recursive traversal (Section 3.1), each loop step may requiring recurring through a subtree before performing the loop step for the next subtree.

Our approach is to divide the problem into two steps. First, we transform an attribute grammar with loops into one without them by unrolling several steps of the loop. Second, after scheduling the loopless grammar, we recover loops from the schedule. Our approach guarantees that, if the synthesizer reports a loopless schedule, dependency-preserving loops will be recovered from it.

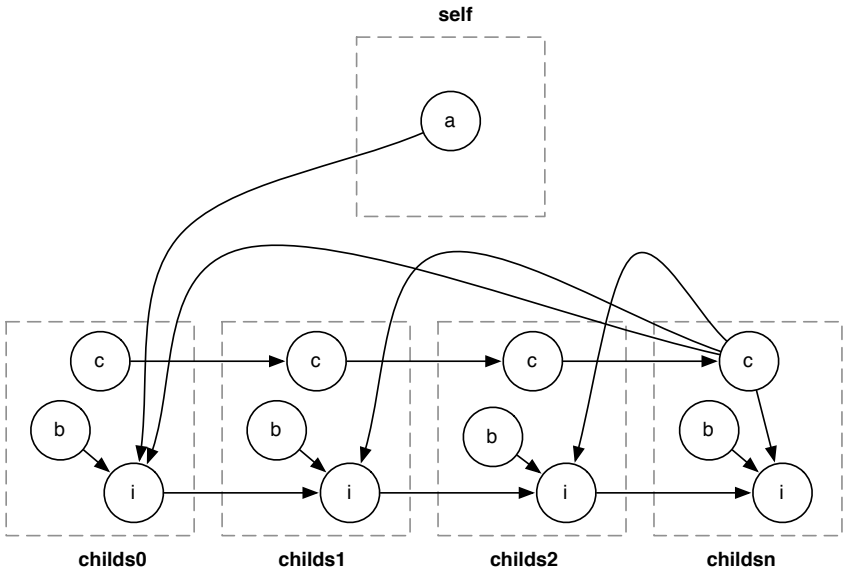
Reduction to OAGs by Unrolling

We show how to schedule the following loop:

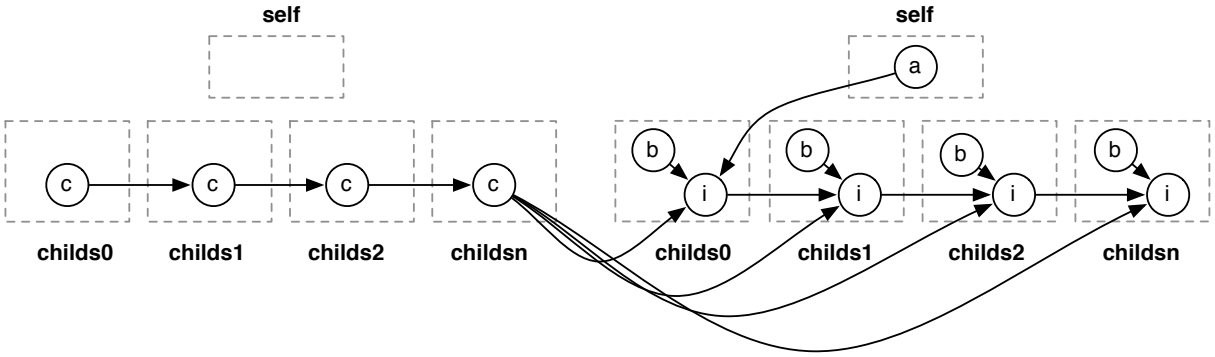
```
1  interface NodeI {
2    var c : int;
3    var i : int;
4    input a : int;
5    input b : int;
6  }
7  class NodeC : NodeI {
8    children {  childs : [ NodeI ]; }
9    actions {
10     loop childs {
11       childs.c := fold 0 .. childs$-.c + 1;
12       childs.i :=
13         fold
14         a
15         ..
16         childs$-.i + childs$i.b + childs$.c;
17     }
18   }
19 }
```

Our reduction unrolls the loop into 4 steps (0, 1, 2, and n):

```
1  interface NodeI {
2    var c : int;
3    var i : int;
4    input a : int;
5    input b : int;
```



(a) **Unrolled Loop Dependencies.**



(b) **Staging as Two Loops.**

Figure 3.8: **Loop scheduling**.The loops may be scheduled for the same traversal if attributes a and b are available.

$$
\begin{aligned}
[\![\ child : [interface]\ ]\!] &\rightarrow child0, child1, child2, childn : interface \\
[\![child.fld := \texttt{fold}\ e_{\texttt{init},fld}\ ..\ e_{\texttt{step}}]\!] &\rightarrow \\
& child0.fld = [\![\ e_{\texttt{step}}[\forall\ f :\ e_{\texttt{init},f}\ /\ child\$\text{-}.f\ ]\!]_0 + e_{\texttt{init},fld}; \\
& child1.fld = [\![\ e_{\texttt{step}}\ ]\!]_1 + child0.fld; \\
& child2.fld = [\![\ e_{\texttt{step}}\ ]\!]_2 + child1.fld; \\
& childn.fld = [\![\ e_{\texttt{step}}\ ]\!]_{\texttt{n}} + child2.fld; \\
[\![\ child\$\$.fld\ ]\!]_\alpha &\rightarrow childn.fld \\
[\![\ child\$i.fld]\!]_\alpha &\rightarrow child\alpha.fld \\
[\![\ child\$\text{-}.fld]\!]_1 &\rightarrow child0.fld \\
[\![\ child\$\text{-}.fld]\!]_2 &\rightarrow child1.fld \\
[\![\ child\$\text{-}.fld]\!]_{\texttt{n}} &\rightarrow child2.fld
\end{aligned}
$$

Figure 3.9: **Rewrite Rules for Loop Reduction**. Cases of $[\![\cdot]\!]$ that simply recur are elided.

```
 6  }
 7  class NodeC : NodeI {
 8    children {
 9      childs0, childs1, childs2, childsn : NodeI;
10    }
11    actions {
12      childs0.c := 0 + 1 + 0;
13      childs1.c := childs0.c + 1 + childs0.c;
14      childs2.c := childs1.c + 1 + childs1.c;
15      childsn.c := childs2.c + 1 + childs2.c;

17      childs0.i := a + childs0.b + childsn.c + a;
18      childs1.i := childs0.i + childs1.b + childsn.c + childs0.i;
19      childs2.i := childs1.i + childs2.b + childsn.c + childs1.i;
20      childsn.i := childs2.i + childsn.b + childsn.c + childs2.i;
21    }
22  }
```

The reduction performs several rewrites that unroll loops and then substitute variable names in the unrolled statements (Figure 3.9). The first key property that the unrolling preserves is that the dependencies are preserved. The unrolling does this in several ways:

- **Schema unrolling.** It unrolls every declaration "child : [NodeI]" into the following form: child0, child1, child2, childn : NodeI
- **Substitution.** The first step of a loop unfolds by replacing references of the form "child$. fld" to use the initial value specified in the first part of a "fold" expression. Likewise, step 1 will substitute the reference with "child0 . fld", and "child$i . fld" for "child1 . fld". Finally, it replaces every reference to last value "child$$. fld" with "child . fld".

- **Forward Loop Direction.** The rewriting enforces a forward loop direction by making child1.fld depend on child0.fld, child2.fld depend on child1.fld, etc. The dependencies simplify later analysis by eliminating concerns in safely reordering steps of a loop. To support an alternative loop order, such as backwards, these dependencies would be elided and the recovery algorithm would be perform more reasoning.

The result of the rewriting is a canonical attribute grammar without loops. Our full implementation differs in two significant ways. First, it performs static checks such as that the fold initialization expression does not refer to step variables "child$i.fld" nor child$.fld. Likewise, statements looping over one collection are checked for references to intermediate elements of another. Second, the rewriting supports loops that may temporarily escape as part of a recursive traversal. Each loop step over an element may require traversal into the element's subtree, so we expand child attributes with a local and transfer version in order to reason about safe placement of the recursive call.

Recovery by Commuting Abstractions

If the rewritten grammar can be scheduled, so can the original grammar with loops. We extract loops from the scheduled grammar and guarantee that any dependency in the original grammar is safely obeyed by the extracted loops. A difficult part of the guarantee is proving that the procedure for recovering loops from the schedule will not get "stuck." This section describes the loop recovery process and its correctness.

The algorithm first rearranges loop statements into distinct blocks. The scheduler may interleave statements from loops over distinct sets of children, but code generation needs them to be separated. Likewise, the algorithm must rearrange all non-loop assignments to fit between loop blocks. The procedure iteratively partitions a sequence of attributes into several subsequences until it reaches a normal form and can proceed nor further.

For each iteration, the algorithm takes a sequences of attributes starting with "child0.fld", which represents the beginning of a loop. It partitions the attributes following it into those that must occur before the loop, after the loop, or during it. The loop's partition is then ready for code generation, and the algorithm repeats on the other two partitions. As the algorithm finalizes a loop with at least one attribute in each step, we guarantee that the algorithm terminates.

The partitions for one step of the algorithm are determined by applying the following three rules:

- **Extract non-loop assignments from a loop**. The beginning of a loop corresponds to an assignment to "child0.fld" and the end by an assignment to "childn.fld". Assignments to non-loop variables that occur within the loop range are moved to be before the beginning of the loop. They are moved out in case multiple statements are scheduled for the same loop and some of them depend on the non-loop variables. All non-loop assignments in a loop range are moved out with their relative ordering preserved.

Moving a non-loop assignment to before the loop is safe. If the assignment depended on a loop variable, that variable could only have been the final one (`child$$.fld`), and the scheduler would not have placed the assignment inside of the loop range. Likewise, if a loop statement depends on the non-loop assignment, moving the assignment earlier preserves the ordering. Finally, moved assignment statements may have mutual dependencies, so maintaining their relative ordering during the movement preserves any read-after-write dependencies.

The process is guaranteed to terminate. First, moving statements out of one loop completes in time linear in the size of the range of the loop. Second, the finite number of loops means that the process only repeat a finite number of times because movement only occurs in one direction.

- **Separate loops over different collections** The algorithm iteratively separates loops over different collections. First, it detects mutually dependent statements that must be scheduled as part of the same loop. Then, it examines the loop span for statements belonging to another type of loop and moves them either to before or after the base range of mutually dependent statements. The moved statements maintain their ordering relative to other statements moved to the same side of the range.

  The complexity of the operation stems from mutually dependent loop variables. For example, the following code must be scheduled into the same loop:

```
1 loop childs {
2   childs.a := fold 0 .. childs$-.b + 1;
3   childs.b := fold 0 .. childs$-.a + 1 + otherChilds$$.c;
4 }
```

  The partial order for the resulting schedule is "(a0|b0) (a1|b1) (a2|b2) (an|bn)". If "cn" is scheduled as "a0 c0 d0 c1 d1 c2 d2 cn b0 dn", the "c" computations do not depend on "a" nor "b" ones, but not vice-versa. The "c" loop must be moved ahead. Doing so is safe relative to "a,b,.." because "c" statements cannot depend on them. Furthermore, if "c" is dependent on other statements in the range, those would also be moved with it, such as seen with "d". For the remaining statements, "a,b,..." do not depend on them and the algorithm moves them after.

  The algorithm terminates because it recursively operates on successively smaller partitions: statements moved earlier, the current loop range, and statements moved after.

- **Separate staged loops over the same collection** Loops over the same collection may still need to be separated. Consider the following loop:

```
1 loop childs {
2   childs.a := fold 0 .. childs$-.b + 1;
3   childs.b := fold 0 .. childs$-.a + 1 + childs$$.c;
4   childs.c := fold 0 .. childs$-.c + 1;
5 }
```

A valid resultant schedule would be the same as the above case. In fact, the same reasoning as applied above applies to this case. For most of the above reasoning, only the loop steps matter.

Due to dependencies across statements being moved before or after a loop, each partitioning step performs all of the above separations. The relative order of statements moved out of a loop is thereby preserved.

Once the partitioning completes, the code generator receives a a list of blocks. Each block is for loop statements or non-loop statements. The code generator handles blocks of non-loop statements as usual. A block of loop statements will translate into a single loop. For example, the above code example code generates as follows:

```
1 for (int i = 0; i < childs.length; i++) {
2         childs[i].c = (i == 0 ? 0 : childs[i - 1].c) + 1;
3 }
4 for (int i = 0; i < childs.length; i++) {
5         childs[i].a = (i == 0 ? 0 : childs[i - 1].b) + 1;
6         childs[i].b = (i == 0 ? 0 : childs[i - 1].a) + 1;
7 }
```

Note that translation of references of the form "childs$-.fld" require tracking the loop step in order to pick whether to use the initial value or the previous node's value.

3.4 Verification

We automatically check an attribute grammar and its schedule for safety. This section focuses on two aspects of our approach: the properties to verify and the modular design of the verification procedure. The properties are significant in that they cover both functional and behavioral correctness, and are typically desired but not proven for layout languages and pattern programs. Furthermore, we check the properties through axiomatic reasoning parameterized by a local dependency analysis. This proof structure simplifies extending the language of statements and of schedules as most additions correspond to an isolated and com posable axiom. Later, in Chapter 4, we show a simple approach to changing the verifier into a synthesizer and thereby achieve fully automatic or computer-assisted parallelization.

Our approach automatically checks three properties.

- **Totality** The specification defines one and only one solution for every well-formed input tree.
- **Determinism** The schedule evaluates the constraints of the attribute grammar without any data races.
- **Linearity (Single Assignment)** Every attribute is assigned exactly once. Layout languages often perform *reflow* to iteratively solve constraints or incremental computation, verifies that reflow is strictly an optimization.

$$\frac{\{A\}\ p\ \{B\} \qquad \{B\}\ q\ \{C\}}{\{A\}\ p\ ;\ q\ \{C\}}\ (seq)$$

$$\frac{\{A\}\ p\ \{B\} \quad \{A\}\ q\ \{C\}}{\{A\}\ p\ ||\ q\ \{B \cup C\}}\ (par)$$

$$\frac{\begin{array}{l} Regions = \{\alpha \mapsto Visit*_\alpha\}\ \cup\ \bigcup_i \{\beta_i \mapsto Visit_i*\} \\ \forall\ \ (\gamma \mapsto Visit*) \in Regions : \\ \quad C_\gamma = \texttt{alwaysCommunicate}_\alpha(\gamma, B, Regions) \\ \quad \{A, C_\gamma\}\ \ \gamma\ Visit*\ \ \{A \cup B_\gamma\} \end{array}}{\{A\}\ \ \alpha\ \ Visit*_\alpha\ \{(\beta_i \mapsto Visit_i)*\}?\ \ \{A \cup\ \bigcup B_\gamma\}}\ (nest_\alpha)$$

$$\frac{\begin{array}{c} P = \cup Prod_i \qquad Steps = \cup Step_j \\ B = \bigcup_i \texttt{reachable}_\beta(Prod_i, P, A, Steps, C) \end{array}}{\{A, C\}\ \ \beta\ (Prod_i\ \{\ Step_j*\ \})*\ \ \{A \cup B\}}\ (check_\beta)$$

Figure 3.10: Correctness axioms for checking a schedule

In this chapter, we illustrate how to check race freedom. The check for linearity is similar , and totality is a consequence of the determinism and linearity properties.

We focus on a modular checking strategy for two reasons. First, we encountered implementation challenges without it. Our initial attempts to adapt the OAG [**oag**] algorithm, which is a search over a global dependency graph, suffered from many implementation bugs before we abandoned it. Our new approach instead decouples verification from synthesis and pattern checking from dependency analysis. Second, challenging our OAG implementation and the basic premise of our approach, we need to support adding new types of traversals. New schedule combinators, such as nested traversals, and individual patterns, such as recursive, should be simple to add as new types of parallel patterns are understood. Adding a parallel pattern should not require refactoring the entire verifier or synthesizer. Our new approach phrases each pattern as an independent axiom and automatically incorporates it into the checking procedure.

$$\texttt{alwaysCommunicate}_{\texttt{parPre}}(\beta, B, M) = \{a_{W,W\to X} \mid (W\to X\ B_\beta) \in M[\beta] \bigwedge_{(V\to W\ B_\gamma)\in M[\gamma\neq\beta]} a_{W,V\to W} \in B \cup A\}$$

(a) Communication check for region boundaries in a **parPre** traversal

$$\begin{aligned}
&\textbf{set}\ \text{reachable}_{\texttt{parPre}}(W\to X, P, A, B, C):\\
&\quad \text{reach} :=\\
&\qquad \{a_{*,W\to X} \mid a_{*,W\to X} \in A\} \cup (C \cap \{a_{W,W\to X} \mid \bigwedge_{V\to W\in P} W.a_{V\to W} \in B\}) \cup (C \cap \{a_{X,W\to X} \mid \neg\exists X\to Y \in P\})\\
&\quad \textbf{while}\ \text{true}:\\
&\qquad \text{progress} := \{a_{*,W\to X} \mid a_{*,W\to X} = f(b_0,\ldots,b_n) \in \mathbf{F}\\
&\qquad\qquad \wedge\ a_{*,W\to X} \in B\ \wedge\ \bigwedge b_i \in \text{reach}\}\\
&\qquad \text{reach} := \text{reach} \cup \text{progress}\\
&\qquad \textbf{if}\ \text{progress} = \emptyset:\\
&\qquad\quad \textbf{break}\\
&\quad \textbf{return}\ \text{reach}
\end{aligned}$$

(b) Unoptimized production visit check for **parPre** traversal

Figure 3.11: Inter- and intra-region checkers for **parPre**.

Axiomatic Checking for Modularity and Correctness

Correctness axioms for checking an entire schedule are in Figure 3.10. The judgements recursively check a composition of traversals until reaching the traversal-specific checks of Figure 3.11. Checking tames worst-case time linear in the number of attributes and the number of their local dependencies. As a reminder, Figure 3.3d defines the language of schedules.

We use a small amount of notation. Variables p and q denote schedules (<Sched>), A and B are sets of attributes, and α and β are traversal types (<travAtomic>). Attribute $a_{W,V\to W}$ is decorated with its production ($V\to W$) and the non-terminal within it (W). We write $a_{*,V\to W}$ if a can be associated with a non-terminal on either side of the production.

The rules to check composition and individual traversals are as follows:

Sequential and parallel composition: ";" and "||" The simplest composition check is for sequencing: Hoare triple "{A} p ; q {C}" (rule **seq**). If attributes A are solved before traversal "p ; q", then attributes C will be solved after. The conditions above the judgement bar state this is true if p can always compute attributes B given attributes A, and q can always then compute C. The judgement is recursive. Analogous reasoning explains "||" (rule **par**).

Nested composition: $\mapsto$ Rule $\textbf{nest}_\alpha$ checks outer traversal type α over regions where each one may have its own traversal type γ. Consider an outer traversal type of **parPre**: as it progresses top-down, every region might be guaranteed to have attributes of its root node

solved before evaluation proceeds within it. For each region (the set of productions mapped to region traversal type γ), the rule calls $\texttt{alwaysCommunicate}_{\texttt{parPre}}$ to find the set C_γ of attributes that are externally set before the region is traversed. Rule $\textbf{nest}_\alpha$ calls checks for every region under the assumption that C_γ is already solved.

The first line of rule $\textbf{nest}_\alpha$ means that, for any outer traversal α, attributes scheduled for the outer region are treated as if they were in their own region ($\gamma = \alpha$). Traversals that do not use nesting are degenerate: all the productions belong to one region ($\gamma = \alpha$).

Traversal over a region (e.g., parPre) The schedule for a traversal of type β over a region is correct if every production visit schedule is correct (rule $\textbf{check}_\beta$). A production visit schedule $Prod_i \ \{ \ Step_j * \ \}$ is correct when there is an order for computing its scheduled attributes $Step_j*$ along which all of the data dependencies of the corresponding semantic functions are satisfied.

Traversals that do not perform nesting, such as a single occurrence of **parPre**, are handled as degenerate nested composition with one region: the entire tree.

Production visit A fast and simple checking algorithm would be to mark each attribute of a production as dirty or clean inside a structure that persists across checks of different visits to the same production. For each successive attribute in a visit's sequence, if all of its dependencies are met (dirty), mark the attribute, and otherwise fail the check. Non-local dependencies can be handled as below.

To optimize the synthesis algorithm of Chapter 4, we use a slightly indirect algorithm to check the correctness of visiting a production. The intuition is that it relaxes the specification of visit's attributes by treating the ordered sequence as an unordered set and checks the reachability of the set's dependency graph.

Figure 3.11b shows an unoptimized reachability computation for visiting a production inside a **parPre** region. It is the standard transitive closure, except for two subtleties:

First, only attributes that are meant to be scheduled are considered reachable (B membership check in line 7). Incorrectly including unscheduled attributes would erroneously allow attributes with unresolved dependencies to also be included.

Second, attributes computed by visits to adjacent productions must be distinguished. Adjacent productions may be in the same region or in another. In a **parPre** region, consider when W is always an intermediate node of the region and attribute $a_{W,W\to X} \in B$ is always set by a parent production $V\to W$ in the same region. For this intra-region case, $a_{W,W\to X}$ is guaranteed to be reachable at the beginning of the visit to $W\to X$. However, if W can be the root node of the region, we must also check $a_{W,V\to W}$ is set by adjacent regions before the root is visited.

Checking an explicit sequence reduces to checking that the transitive closure can be performed in the specified order rather then the declarative definition in Figure 3.11 . The synthesizer of Chapter 4 does not need to check for ordering, so we omit this check.

Property Proofs

The axioms check for determinism, which can be adapted to check the two other properties.

First, the axioms check determinism, which means that rerunning the schedule will yield the same result. We can check determinism by ensuring that a schedule computes the attributes of an attribute grammar without races. More precisely, it tracks what attributes are guaranteed to have been computed by any particular point of the schedule, and uses that to check that every step of the computation only relies on what is guaranteed to have been computed.

Linearity requires that that every instance of an attribute is only assigned to once. We can check linearity by extending the axioms in two ways. First, they must check that for any given attribute X_a, it is either defined by all productions $X \to W$ or by all productions $W \to X$. Second, for every attribute assigned in production $X \to W$, it must only be scheduled for one visit.

Totality guarantees that every well-formed input tree yields one and only one result. It is a property of the language because any schedule must reach the same result. In contrast, determinism is a property of a schedule because, for the same language, rerunning one schedule may always return the same result while the same might not be guaranteed for another schedule. The proof of totality lies in the proof of linearity. Given a linear schedule, the dynamic dependency graph of every input document is directed and acyclic. The DAG property guarantees that the value of each node is a pure function of the values of the dominating nodes, and therefore the language has a (total) functional interpretation. Checking totality adds an additional step beyond checking linearity: totality requires that every attribute in the grammar appears in the schedule.

Verification is $O(|A|)$

Verifying a schedule for race-freedom takes time linear in the number of attributes. Our description of the checker in Figures 3.10 and 3.11 does not show the optimizations that lead to this bound so we highlight the key ideas here.

First, the number of axioms to check is linear in the number of attributes. Every traversal computes at least one attribute, which implies that the number of traversals is bounded by the number of attributes. Only non-empty visits must be checked, and their number is likewise bounded by the number of attributes.

Second, the time to check an axiom is linearly bounded by the number of attributes (and their dependencies) to be scheduled by that axiom. For example, checking the visit to a production is effectively a topological sort of the local dependency graph restricted to the attributes evaluated during the visit. Topologically sorting dependency graph $G = (E, V)$ is $O(|E|+|V|)$. An attribute can have at most $|A|$ local dependencies so verification takes time $O(|A|)$. For simplicity, our implementation does not use the topological sort optimization, and we only encountered performance issues on one case study due to that.

Similar reasoning applies to quickly checking the two other properties. Linear checking of linearity follows the same proof structure. Each attribute is labeled based on the type of production that solves it and rule $\textbf{check}_\beta$ checks that the labels of attributes in $Step_j$ match production $Prod_i$. The time to check the axiom is therefore still bounded by the number of scheduled attributes. Finally, totality checks that the computed set of attributes matches the total set, and comparing two sets is also linear in the number of attributes.

3.5 Evaluation: Layout as Structured Parallel Visits

We show that our static language of parallel schedules is expressive enough to support common layout tasks. In particular, we show that it can support document layout (box models and nested text), table layout (user interfaces and data tables), and rendering. Our document layout and table examples describe supporting a subset of CSS. Our rendering example highlights optimizing dynamic memory allocation for GPU. Rendering should be integrated into each layout model; we discuss how to do so at the end. The attribute grammars in Appendix **??** include sketches 4 of the schedules described here.

Box model

Document languages provide nested box models where intermediate nodes are boxes and leaf nodes are text and images. For example, a box may represent a page, column, or paragraph. The `H-AG` example provides the basic insight, except a language such as CSS extends it with features. Of most relevance to parallelization, we describe supporting the following common features with non-trivial data dependencies [**mama**]:

- Intrinsic preferences. Document content leads to intrinsic preferences, such as as a box being big enough to contain its content. These must be combined with external constraints, such as overriding preferences set by the designer on the element or its container.
- Relative layout. Based on the size preferences of a node and its content, the content must be positioned relative to each other and the node.
- Absolute layout. Based on the relative positioning between a node and its parent, transitive reasoning must be applied to position the node relative to the tree's root node.

Our static box model schedule loosely corresponds to the above list by devoting 1-2 parallel passes for each item.

We stage the computations with the following sequence of parallel traversals:

1. **Bottom-up: intrinsic widths and concrete overriding constraints.** For example, the intrinsic width of a horizontal box is the sum of intrinsic widths of its children. If the user specifies a concrete width value such as 2 pixels, that value is used instead.

2. **Top-down: percent widths.** Constraints such as a width being a percent of its parent are computed next. Notably, the CSS standard defines percent widths that cannot be computed at this point as being undefined. The definition by the CSS standard makes whatever interpretation we use safe.

3. **Bottom-up: heights and relative positioning.** Once the size of a node's children is known, their placement relative to the node can be computed. For example, a horizontal box would place them side by side, and a vertical box would stack them. Likewise, the relative positioning of a node's children, their heights, and any overriding user constraints are sufficient for computing the node's height.

4. **Top-down: absolute positioning.** When the absolute position of a node becomes available, the absolute positions of its children may be computed. The process proceeds recursively.

Nested text

Our core approach to supporting nested text is described in Section 3.1. As a reminder, the problem is to perform word-wrapping on subtrees such as paragraphs with stylized text. The idea is to identify subtrees that require sequential evaluation but can be computed in parallel with other subtrees. Given the basic insight of performing such a nesting, we use our tool to design and verify the schedule.

A non-obvious aspect of using nested traversals for text layout is that we only use the nesting for one pass. The computations relating to text layout span several tree traversals. The intrinsic and computed width passes execute using the parallel traversals described above. We only use the nesting for height and relative position computation. Thus, our strategy of using nested traversals achieves coarse-grained parallelism for the traversal with the difficult word-wrapping dependency, and features the usual fine-grained parallelism for all others.

Grids

We scheduled the automatic layout algorithm used in CSS and HTML as parallel tree traversals. Section 2.4 describes the functional specification. The primary dependencies challenging parallelization relate to supporting topological traversals over a DAG rather than a tree because cells have two parents: the row and the column. In a top-down traversal, both the row and column should be visited before the cell. Our solution for parallelization is a level-synchronous breadth-first evaluation order. Finally, our example propagates information between rows and columns using several intermediate parallel traversals. A nested traversal or the ability to reason about attributes of grandchildren rather than just children may help eliminate those traversals.

SIMD Rendering through Staged 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.

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 3.1: 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 3.1 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 3.6, 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

Discussion

An interesting distinction arises between our use of attribute grammars and our approach of parallel traversals. The restricted attribute grammars prevents specifying computations that

could be manually expressed with the tree traversals. Thus, we found that our box model can be manually encoded with 3-5 parallel traversals, but expressing it with our restricted attribute grammar formalism may require additional parallel traversals. An important future direction would therefore be to increase the flexibility and reasoning over the attribute grammar while maintaining the set of parallel traversal types.

3.6 Related Work

Lang of schedules

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

Schedule verification

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

Chapter 4

Parallel Schedule Synthesis

Programmers struggle to map applications into parallel algorithms. Going beyond the automatic schedule verification of the last chapter, we now examine how to automatically generate a schedule. Consider two of the decisions that a programmer faces in manually designing a schedule:

- **Scheduling a single traversal.** Many computations contain sequential dependencies between nodes. One correct traversal over the full tree might then be sequential. However, if the sequential dependencies can be isolated to a subtree, an overall parallel traversal would be possible if it invokes a sequential traversal for just the isolated subtree. Whether such isolation is always possible is not obvious.
- **Scheduling multiple traversals.** Programs such as browsers perform many traversals. Traversals might run one after another, concurrently, or be fused into one. These choices optimize for different aspects of the computation. Running two traversals in parallel improves scaling, but fusing them into one parallel traversal avoids overheads: the choice may depend on both the hardware and tree size. Which traversal sequence to use is not obvious.

These decisions explode the space of schedules. Today, programmers manually navigate the space by selecting a parallel schedule, judging its correctness, and comparing its efficiency to alternative schedules. The tasks are expensive: programmers globally reason about dependencies, develop prototypes for profiling, and whenever the functional specification changes, restart the process.

This chapter explores the design and implementation of an attribute grammar that supports automatic schedule synthesis. We examine several questions:

- What programming constructs are enabled by schedule synthesis?
- What is an algorithm to *quickly* find a *correct* schedule?
- If multiple schedules are possible, how do we find a *fast* one?

The following sections explore each question in turn.

4.1 Computer-Aided Programming with Schedule Sketching

Automatic parallel schedule synthesis enables rich forms of parallel programming. The utility of these constructs is not obvious. An automation tool will automatically find a parallel schedule, so a natural conclusion would be to assume the programming interface simply hide all parallelization concerns and rely upon automatic parallelization internally. We found this to be largely true for writing small amounts of declarative data visualization code. However, in parallelizing the larger and more complicated CSS layout language, we encountered cases where the visualization designer needed to guide (or be guided by) the automation procedure. Likewise, we encountered the need for one programmer to communicate parallel structure to another. Automatic parallelization is insufficient in that it hides all parallelization details and controls, yet manual scheduling was too low-level and brittle.

Our solution is to provide a *sketching* construct for specifying constraints on the schedule that the automatic parallelization algorithm must respect. The programmer chooses which parts of a schedule to write and relies upon the synthesizer to fill in the rest. We routinely sketched schedules in order to *override schedule selection*, *test* and *debug* parallelization ideas, and *enforceably communicate parallelization decisions* when sharing code with others. Discussed later in this chapter **??**, we also used the sketching mechanism to speed up automatic parallelization over specifications with many attributes or schedule patterns that challenge static analysis.

We revisit the specification of `H-AG` to demonstrate the sketching construct and its use for the above scenarios. First, depending on the expected memory size of target hardware, the programmer may choose a longer schedule with a smaller set of attributes computed in each one. Compare the three following schedule sketches:

$$
\begin{aligned}
&\mathit{?hole}_1 \\
&\textbf{parPost}\ \mathit{?hole}_2\ ;\ \textbf{parPre}\ \mathit{?hole}_3 \\
&(\textbf{parPost}\ \mathit{?hole}_4\ ;\ \textbf{parPost}\ \mathit{?hole}_5)\ ;\ \mathit{?hole}_6
\end{aligned}
$$

The first specification leaves a *hole* for the entire schedule. The synthesizer fills in every hole with a valid schedule term so that the resulting schedule is correct. The entire first schedule is left as a hole, which is equivalent to requesting fully automatic parallelization. The second specification hardcodes the traversals but leaves holes for the attributes to schedule for each traversal. The final schedule sketch splits the parPost traversal in two in order to decrease the memory consumption in the first traversal. Like the second sketch, it does not specify the attributes, and like the first sketch, it does not specify the sequence of traversals to place at the end of the schedule.

The ability to run a sketch through the synthesizer enables several forms of parallel program debugging. First, the synthesizer rejects programs that it cannot parallelize, so sketches can test programmer intuitions. For example, it could test the validity of the above idea of splitting apart the first parPost traversal. We could more explicitly test the underlying insight that the w and h attributes are separable:

$$(\textbf{parPost}\ \{w\}\ ;\ \textbf{parPost}\ \{h\})\ ;\ ?hole_6$$

The synthesizer can fill in $?hole_6$ to find a complete schedule. It provides an error if it cannot: the longest schedule prefix of traversals it could schedule. For the above example, the error distinguishes two possible mistakes. First, it fails with a prefix containing parPost { w } ; parPost { h }, the first traversal can be split but the rest of the schedule has an unsatisfiable dependency. Otherwise, the prefix is empty and the traversals could not be split. We found the ability to test scheduling ideas to be particularly useful, for example, in determining partitions for nested text.

We provide another mechanism for debugging. The programmer may ask the synthesizer to *enumerate* all valid solutions for a schedule sketch. The previous examples restricted themselves to only asking for one completion. However, for `H-AG` , the space of valid schedules is small enough that programmer could manually page through all possibles ones.

As our attribute grammars grew, we wrote sketches to help share code between programmers. Consider a program with a sketch such as the above. Upon receiving a grammar with it, a programmer knows the desired parallelization scheme. Furthermore, the synthesizer checks that edits to the functional specification do not violate the schedule. For example, the synthesizer would detect the addition of a feature that requires the addition of an extra traversal or serialization of a parallel one. We typically ignored changes that do not impact parallelization and applied more careful reasoning whenever a sketch was violated. In this way, the ability to communicate and enforce schedule specifications helps separate concerns between defining layout feature logic and optimizing layout scheduling.

4.2 Generalizing Holes to Syntactic Unification

We provide a more expressive variant of holes for cases that require more high-level control than they provide. For example, we may want to specify that both the width and height are computed in the first traversal over a tree. The programmer should not have to specify the relative order of attributes for every type of node that computes them. Instead, we generalize the sketching construct to syntactic unification over scheduling terms.

Programmers may specify constraints over schedule terms. For example, the following specification declares that the width and height attributes are computed in the first traversal

of a sequence but do not specify their relative order:

$$
\begin{aligned}
&\mathit{member}(\text{w}, \mathit{?hole}_2), \mathit{member}(\text{w}, \mathit{?hole}_3),\\
&\mathit{member}(\text{h}, \mathit{?hole}_2), \mathit{member}(\text{h}, \mathit{?hole}_3),\\
\textbf{\textit{Sched}} = \ &[\ [\mathit{?hole}_1,\ [[\text{HBox}, \mathit{?hole}_2], [\text{VBox}, \mathit{?hole}_3]]],\\
&\ \ \textbf{seq},\\
&\ \ [\textbf{parPost},\ \mathit{?hole}_4]\ ]
\end{aligned}
$$

Term $\mathit{?hole}_1$ will unify with a traversal type and $\mathit{?hole}_2$ and $\mathit{?hole}_3$ will unify with a sequence of attributes that includes w and h. Finally, $\mathit{?hole}_4$ will unify with another sequence of terms where each specifies a node type and the sequence of attributes to schedule for it. Note the change in syntax.

Our scheduling language is an embedded domain specific language (EDSL [**??**]) in Prolog. The language of constraints is arbitrary Prolog. Thus, in the above example, Sched is a named Prolog variable that must be unified with the schedule constraints and the attribute grammar's functional dependencies. Likewise, unnamed variables $\mathit{?hole}_1$, $\mathit{?hole}_2$, and $\mathit{?hole}_3$ must unify with a correct schedule. Our system provides a library of traversal types such as parPost and combinators such as seq, and the attribute grammar introduces attribute terms such as w and h. The programmer then uses built-in Prolog predicates to constrain the result such as member for list membership. Likewise, they may use Prolog's "," operator for conjunction and ";" for disjunction.

We made several notable uses of the extended sketching constructs:

- **Attribute sets.** As in the above example, we specify *unordered sets* of attributes for a traversal rather than an *ordered sequence*. The synthesizer determines the order and any additional attributes. Likewise, different classes implementing the same interface share attributes of the same name, we wrote helper functions for expanding an interface attribute to schedule into the instances for the corresponding classes.

- **Requiring parallelization.** We may specify that a traversal type unifies with a parallel form:

$$
\begin{aligned}
&(\mathit{?hole}_1 = \textbf{parPost}; \mathit{?hole}_1 = \textbf{parPre}),\\
&\textbf{\textit{Sched}} = [\ [\mathit{?hole}_1,\ \mathit{?hole}_2],\ \textbf{seq},\ [\textbf{parPost},\ \mathit{?hole}_3]]
\end{aligned}
$$

  The sketch specifies a sequence of two traversals where the first traversal type ($\mathit{?hole}_1$) unifies with either parPost or parPre traversal. The schedule does not specify what attributes are computed within the first traversal ($\mathit{?hole}_2$). Furthermore, instead of manually specifying the choice for every pass, we wrote a function that does so automatically.

- **Nesting.** We use predicates to test the validity of partitioning into nested traversals. The challenge is to minimize the number of traversals put into a sequential partition.

For example, if we thought a node belongs in a parallel partition, we would include that in the sketch;

$$member([\text{HBox}, ?hole_1], TopDownVisits),$$
$$\textbf{\textit{Sched}} = [\ [\textbf{nested}, [\textbf{parPre}, TopDownVisits],\ ?hole_2]\ |\ ?hole_3]$$

The schedule specifies that the first traversal is nested within an overall parallel preorder structure. The preorder portion must handle HBox nodes and may include other as well. The other partitions are defined by $?hole_2$, which has no additional constraints. The specification leaves remaining traversals unconstrained by $?hole_3$.

- **Schedule heuristics.** For a simple optimization heuristic, we would bias parallel scheduling to use an alternating sequence of parPre ancd parPost traversal. The intuition is that long-running dependencies can often be satisfied under these two traversals so that the shortest schedule would be such an alternation. Less obviously, some dependencies require repetition of the same traversal pattern, so the heuristic is to bias to use of the alternate of whatever worked for the previous traversal and otherwise prioritize any other parallel traversal.

In all of the above cases, reasoning is in terms of the syntactic form. For example, the alternating traversal heuristic biases towards one traversal based on equality with the syntactic value of the the previous one. Richer forms of unification that extend beyond syntax may be applicable. As is, syntactic unification for guiding schedules already supports several key tasks.

4.3 Fast Algorithm for Schedule Synthesis

Our synthesizer takes an attribute grammar and a sketch as input, and outputs a set of schedules. We designed it to support multiple traversal types, multiple solutions, and rich attribute grammar and schedule sketching languages. Our initial implementation used the dependency analysis of **oag** [**oag**], but it was too inflexible. Our new algorithm optimizes for modularity and speed by using the following design:

Simple enumerate-and-check The algorithm enumerates schedules and checks which are correct. Checkers examine the use of individual traversal types and traversal compositors, and we wrote them to function independently of one another. Enumeration is simply syntactic. Combined, adding a new traversal type involves writing a checker and binding it to the proper place.

Optimization Naïve enumerate-and-check is too slow. Without significantly changing the interface for adding checkers, we optimize synthesizing one schedule to be $O(n^3)$ in the number of attributes. Some features are still slow, such as nested traversals, so we introduce the optimizations of incrementalization, greediness, and greedy sketch unification.

```
 1  parPre{x,y,w,h}                                  incorrect: unsat {x,w,h}
 2  parPre{y}                                        correct: continue
    ... /* expand subtree to schedule x, w, h */ ...
 3  parPost{x,y,w,h}                                 incorrect: unsat {x,y}
 4  parPost{w,h}                                     correct: continue
 5    _ ; parPre{x,y}                                correct: complete
 6    _ ; parPost{x,y}                               incorrect: unsat {x,y}
 7    _ ; (parPre{x} || _)                           correct: continue
 8      _ ; (_ || parPre{y})                         correct: complete
 9      _ ; (_ || parPost{y})                        incorrect: unsat {y}
10    _ ; (parPre{y} || _)                           correct: continue
11      _ ; (_ || parPre{x})                         correct: complete
12      _ ; (_ || parPost{x})                        incorrect: unsat {x}
13    _ ; (parPost{y} || _)                          incorrect: unsat {y}
14    _ || parPre{x,y}                               incorrect: unsat {x}
15    _ || (parPre{y} ; _)                           correct: continue
16      _ || (_ ; parPre{x})                         incorrect: unsat {x}
17      _ || (_ ; parPost{x})                        incorrect: unsat {x}
    ...
18  parPost{w}                                       correct: continue
19    _ || parPre{x,y,h}                             incorrect: unsat {x,h}
    ...
```

Figure 4.1: **Trace of synthesizing schedules for** `H-AG` . Note that scheduling of "||" does not use the optional greedy heuristic.

The Algorithm

We first discuss optimizations for finding one correct schedule before considering finding many. Figure 4.1 demonstrates an algorithm trace for enumerating schedules of `H-AG` . Figure 4.2 shows the full algorithm.

Synthesizing one schedule is $O(A^3)$ in the number of attributes. The algorithm finds an increasingly long and correct prefix of the schedule (*prefix expansion*). At each step, it tries different suffixes until one succeeds, where a suffix such as "**parPre**{x,y}" is a traversal type and attributes to compute in it. When a correct suffix is found, it is appended to the prefix and the loop continues on to the next suffix. Finding one suffix involves trying different traversal types, and for each one, different attributes. Only the suffix needs to be checked (*incremental checking*), and checking a suffix is fast (*topological sort*). Finally, finding a set of attributes computable by a particular traversal type only requires $O(A)$ attempts (*iterative refinement*).

We consider each optimization in turn:

1. **Prefix expansion.** The synthesizer searches for an increasingly large *correct* schedule prefix. Every line of the trace represents a prefix. If a prefix is incorrect, no suffix will yield a correct schedule. Therefore, the only prefixes that get expanded are those that succeed (lines 2, 4, 7, 10, 15, 18).

   To synthesize only one schedule, only one increasingly large prefix is expanded. Line 2 has a correct prefix, so only "**parPre**{y}" would be explored. Either no schedule is

possible at all, or if there are any, one is guaranteed to exist in the expansion. In this case, "**parPre**{y} ; **parPost**{w,h} ; **parPre**{x}" would be found.

2. **Incremental checking.** Line 4 checks prefix "**parPost**{w,h}" for attributes "w" and "h." Therefore, lines 5-17 can check the suffix added at each line without rechecking "**parPost**{w,h}".

3. **Topological sort.** We optimize checking a suffix by topologically sorting the dependency graph of its attributes (rule $\textbf{c}heck_\beta$ in the next subsection). Topologically sorting a graph is $O(V + E)$. It is $O(A)$ in this case because $V = A$, and as the arity of semantic functions is generally small, E is $O(A)$.

4. **Iterative refinement.** The algorithm iteratively refines an over-approximation of what attributes can be computed in a suffix by removing under-approximations of what cannot. For example, the check in line 1 for **parPre**{x,y,w,h} fails with error {x,w,h}, which details the attributes with unsatisfiable dependencies. Computing fewer attributes cannot satisfy more dependencies, so no subset of {x,w,h} has satisfiable dependencies either. Therefore, the next check is on a set without them: {y}.

   Subtraction of attributes repeats at most A times before finding a solution or terminating on the empty set. Checking one refinement invokes the $O(A)$ topological sort. Put together, finding the attributes computable by a suffix is $O(A^2)$.

Every traversal computes at least one attribute, so there are at most A traversals. A constant number of traversal types are examined for each suffix, and synthesizing each one is $O(A^2)$. Synthesizing one schedule is therefore $O(A^3)$. The *greedy sketch unification* optimization from the next section may further optimize the synthesis of a single schedule.

4.4 Schedule Enumeration

We provide and optimize the ability to examine many schedules. Our approach benefits several scenarios: picking a fast schedule when many are possible, supporting scheduling language extensions that otherwise resist fast synthesis, and improving synthesis time when partial schedule knowledge is known.

We consider each scenario in turn:

Autotuning There may be an exponential number of schedules, and the choice of the fast is non-obvious. For example, shorter schedules incur less traversal overhead, but also generally expose less parallelism. Likewise, a short sequence of parallel traversals may behave worse than a long sequence when performed on hardware with limited memory. By enumerating all schedules, we can perform *autotuning*: run performance tests to pick the best schedule for a particular architecture. There may be an exponential number of schedules, so we must somehow optimize the enumeration of those to test.

Scheduling extensions. We provide optional scheduling language extensions, and fast synthesis in their presence requires optimization. For example, nested traversals require partitioning the set of nodes into distinct regions, but many partitions are possible. Partitioning does not enjoy the monotonicity property that we previously exploited and thus, on its own, is slow.

Faster synthesis. Verification is linear-time in theory yet running the synthesizer to perform it, as is, takes cubic time. If the programmer provides knowledge of the schedule, such as when recompiling the grammar, synthesis should execute faster. In the limit, providing full schedule knowledge should reduce synthesis time to that of verification.

We introduce several optimizations that, together, address the above scenarios. They optimize for when multiple schedules may be valid schedules, and except for backtracking, may also improve the process of finding one schedule.

- **Backtracking.** To emit multiple schedules, we extend prefix expansion to also perform backtracking. After a schedule is fully completed or a suffix fails, the synthesizer backtracks to the most recent correct prefix. For example, line 8 is a complete and correct schedule. Backtracking returns to the earlier correct prefix of line 7 and tries the alternative suffix of line 9.

- **Greedy sketch unification.** We use sketches to prune the search. For example, sketch

  parPost *?hole*$_1$ || *?hole*$_2$

  enables skipping lines 1-3 because they do not start with a **parPost** traversal. Lines 5-13 could also be skipped because the compositor is not "||".

  A sketch that provides a full schedule reduces synthesis to verification, which is $O(A)$ (topological sort). Sketching also enable features that otherwise require exponential search to still synthesize in $O(A^3)$. For example, scheduling nested regions is exponential in the number productions, but if just the production partitioning is sketched, synthesis for the remaining schedule terms is still only $O(A^3)$.

- **Greedy attribute heuristic.** For any schedule "p ; q", solving fewer attributes in p will not enable solving q with fewer traversals. Thus, to minimize the number of traversals, all such subsets are pruned. For example, as line 4 found **parPost**{w,h}, line 19 skips "**parPost**{w} ; _ " and proceeds to "**parPost**{w} || _".

  Greediness reduces enumerating all schedules to only being exponential in the number of traversals. This is significant because, for example, our schedule for CSS has only 9 traversals.

```
1  def synthFast(sketch):
2    yield synth(∅, Attributes, sketch)

4  def synth(prev,rest,sketch):
5    choose  ⊗ ∈ { ";", "||" }
6    if ⊗ = ";":
7      choose α ∈ { "parPre", "parPost", ... }
8      A := iterativeRefine(α, prev, rest)
9      if A = rest:
10       unify(sketch, α A)
11       yield α A
12     else if A = ∅:
13       backtrack
14     else:
15       unify(sketch, α A ; rhs1)
16       yield α A ; synth(prev ∪ A, rest − A, rhs1)
17   else:
18     unify(sketch, lhs2 || rhs2)
19     choose A ⊂ rest
20     p := synth(prev, A, lhs2)
21     q := synth(prev, rest   A, rhs2)
22     yield p || q

24 def iterativeRefine(α, prev, rest):
25   overapproxA = rest
26   do:
27     X = check_α(prev, overapproxA)
28     overapproxA = overapproxA − X
29   while X ≠ ∅
30   yield overapproxA
31   if nonGreedy:
32     choose overapproxA′ ⊂ overapproxA
33     yield iterativeRefine(α, prev, overapproxA′)
```

Figure 4.2: **Optimized synthesis algorithm.** Lines 10,15,18: early unification with sketches. Lines 8,27: incremental checking. Line 26: iterative refinement. Line 31: toggle minimal length schedules. Lines 12,28: pruning of traversals with unsatisfiable dependencies.

In summary, synthesizing one schedule in our base language is $O(A^3)$, but emitting all of them is exponential. Likewise, scheduling language extensions such as nested traversals still support fast synthesis of surrounding terms when guided by sketches. Our optimizations optimize the process, such as by reducing synthesis complexity to that of verification when increasingly detailed sketches are provided.

4.5 Evaluation

We evaluate the automation capabilities of our schedule synthesizer for our case studies of data visualization and document layout. First, we examine whether the synthesizer could find parallelism and how much guidance it needed. Second, we evaluate whether our synthesis algorithm can achieve interactive or same-day compile times. Finally, we examine the quality

name	loc	1st	sketch	found	avg
hbox++	305	5.6s	9.6s	54	2.7s
spiral	144	0.7s	0.9s	12	0.4s
votes	327	15.4s	22.0s	36	8.0s
css	1132	1919.6s	65.1s	100	445.4s

Figure 4.3: **Synthesizer speed:** `1st` is the time to first schedule without using a sketch. `sketch` is the time to first schedule using a sketch of the traversal sequence. `found` is the number of schedules found. `avg` is the average time to find a sketch.

of schedules: we measure the benefit of autotuning and the cost of our greedy heuristic.

Automatic Parallelization

We first evaluate whether the synthesizer automatically detected parallelism and the amount of schedule guidance we provided.

For all of the data visualizations (tree map, single and multiple time series, hbox, and sunburst), we successfully relied upon the synthesizer to automatically find parallelism. We performed an iterative design process where we would iterate between adding a feature and checking that the compiler could automatically parallelize it. Once the compiler accepted one functional specification, we would extend the specification with the next feature. When satisfied with the visualization, we would specify the sketch of parallel traversals, but only for communicating requirements to future programmers.

The CSS specification required guidance. On its own, the synthesizer would find a sequence of parallel preorder and postorder traversals. The exception is one traversal that requires nested partitions for parallelization, so to improve synthesis times, we specified the structure of that traversal. Furthermore, due to the many cross-cutting data dependencies in CSS, we specified schedule sketches throughout the design process. The sketches ensured that extensions to the functional specification did not violate our understanding of the parallel behavior.

Figure 4.3 shows the lines of declarative code for each specification. The generated code was over a magnitude more depending on the compiler backend. The number of parallel traversals ranged from 3 to 9.

Synthesis Speed

Performing synthesis in less than a minute enables interactive use by programmers, and even faster times would support runtime compilation. We measured the time to synthesize several attribute grammars. Figure 4.3 shows the lines of code for each one and various timings on a 2.66GHz Intel Core i7 with 4GB of RAM.

Generally, synthesizing a schedule, whether an arbitrary one (`1st`) or from a sketch specifying the traversal sequence (`sketch`), takes less than 30 seconds. The exception was CSS, which we discuss later and was still fast.

Emitting all schedules is even faster per emitted schedule (`avg`) than just finding the first. While the total time to emit all schedules can be slow, we note that enumeration is for offline autotuning. Finally, the greedy heuristic was necessary for enumerating schedules. Even after one day of running the non-greedy algorithm for CSS, most of the greedy CSS schedules were still not emitted.

Overall, we see that synthesis is fast enough for interactive use by the programmer.

Autotuning

We evaluated schedule autotuning speedups for `hbox++` on `laptop`:

Comparing greedy schedules We enumerated greedy schedules for `hbox++` and compared performance on 1 and 2 cores. The relative standard deviation for performance of different schedules (σ/μ) is 8%. The best schedules for 1 and 2 cores are different. Swapping them leads to 20-30% performance degradation, and the difference between the best and worst schedules for the two scenarios are 32% and 42%, respectively. Autotuning schedules improves performance.

Comparing greedy to non-greedy Our schedule enumeration is not exhaustive because of the greedy heuristic, and therefore may miss fast schedules (Section **??**). For a fixed schedule of traversals with a greedy attribute schedule, non-greedy attribute schedules were 0-6% faster. On average, however, non-greedy schedules were 5% slower. Greedy scheduling was safe for `hbox++` .

In our case studies, much of the benefit of autotuning derives from trying to greedy schedules: one that starts with a parallel preorder traversal and then alternatives with parallel postorder ones, and vice-versa. We did see exceptions however, such as CSS benefiting from a nested traversal, and some grammars requiring occasional repetition of preorder and postorder traversals.

Chapter 5

Optimizing Tree Traversals for MIMD and SIMD

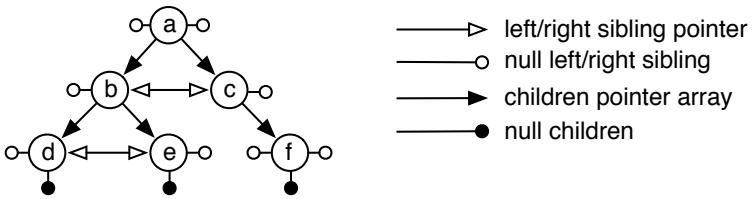
5.1 Overview

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 two ways:

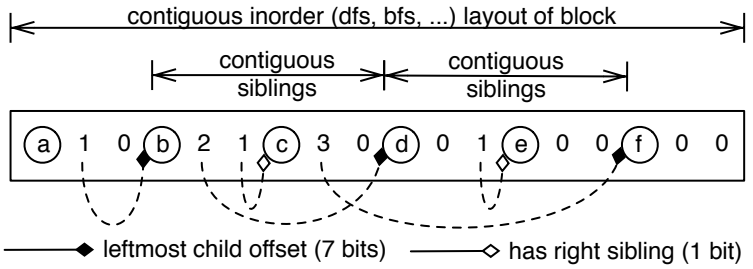
1. **Semi-static work stealing for MIMD:** MIMD traversals should be optimized for low overheads, load balancing, and locality. Existing techniques such as work stealing provide spatial locality and, with tiling, low overheads. However, dynamic load balancing within a traversal leads to poor temporal 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.

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.



(a) Naïve pointer-based tree representation



(b) Compressed tree encoding

Figure 5.1: Two representations of the same tree: naıve pointer-based and optimized. The optimized version employs packing, breadth-first layout, and pointer compression via relative indexing.

5.2 MIMD: Semi-static work stealing

We optimize the tree data representation and runtime schedule for MIMD evaluation. We did not see significant parallel speedups when either one was left out. Through a non-trivial amount of experimentation, we found an almost satisfactory combination of existing techniques. It includes popular ideas such as work stealing [[CITE]] for load-balanced runtime scheduling and tiling [[CITE]] for data locality, so we report on how to combine them. However, we did not see more than 2X speedups until we added a novel technique to optimize for low run-time scheduling overheads and temporal data locality: semi-static work stealing. The remainder of this section explores our basic data representation and runtime scheduling techniques.

Data representation: Tuned and Compressed Tiles

Our data representation optimizes for spatial and temporal locality and, as will be used by the scheduler, low overheads for operating over multiple nodes. Many researchers have proposed individual techniques for similar needs, and it is unclear which to use for what hardware. For example, mobile devices typically have smaller caches than laptops, they should exchange time for space. Our solution was to implement many techniques and build an autotuner [[CITE]] that automatically choose an effective combination.

Our autotuner runs sample data on multiple configurations for a particular platform to

decide which configuration to use. The most prominent options are:

- C++ collections or contiguous arrays
- tiling [**tiling**] of subtrees
- depth-first or breadth-first ordering of nodes in a tile (with matching traversal order [**Chilimbi:1999**])
- aligned data, or unaligned but more packed data
- pointer compression

Several of the techniques are parameterized, so our tuner performs a brute force search for parameter values such as the maximum size of a subtree tile. To make the search tractable, we prune by manually providing heuristics, such as for parameter ranges.

The individual optimizations target several objectives:

- **Compression** Compressing the tree better utilizes memory bandwidth and decreases the working set size. We use two basic techniques: structure packing and pointer compression. Packing combines several fields in the same word of memory, such as storing 32 boolean attributes in one 32bit integer field. Similar to **compression** [**compression**], compression encodes node references as relative offsets (16–20bits) rather than 32bit of 64bit pointers. Likewise, as there are typically few siblings, instead of a counter of number of children (or siblings), we use an isLastSibling bit. Figure 5.1 depicts a tree using pointers and one of our representations: in the example, the compressed form uses 96% fewer bits on a 64-bit architecture.

- **Temporal and Spatial Locality** The above compression optimizations improve locality by decreasing the distance between data. To further improve locality, we support rearranging the data in several ways .

  Tiling [**tiling**] cuts the tree into subtrees and collocates nodes of the same subtree. It improves spatial locality because a node only reads and writes to its neighbors. Likewise, we support breadth-first and depth-first node orderings within a subtree (and across subtrees). Such a representation matches the tree traversal order [**Chilimbi:19999**] and therefore improves temporal locality.

- **Prefetching** We supports several options for prefetching to avoid waiting on data reads. First, the data access patterns with the data layout, so hardware prefetchers might automatically predict and prefetch data. Second, our compiler can automatically insert explicit prefetch instructions as part of the traversal. Finally, runahead processing [**runaheadprocessing**] pre-executes data access instructions. A helper thread traverses a subtree ahead of a corresponding evaluator thread, requesting node data while the evaluator is still computing an earlier thread. We only saw benefits of the first in practice, but leave the others as tunable.

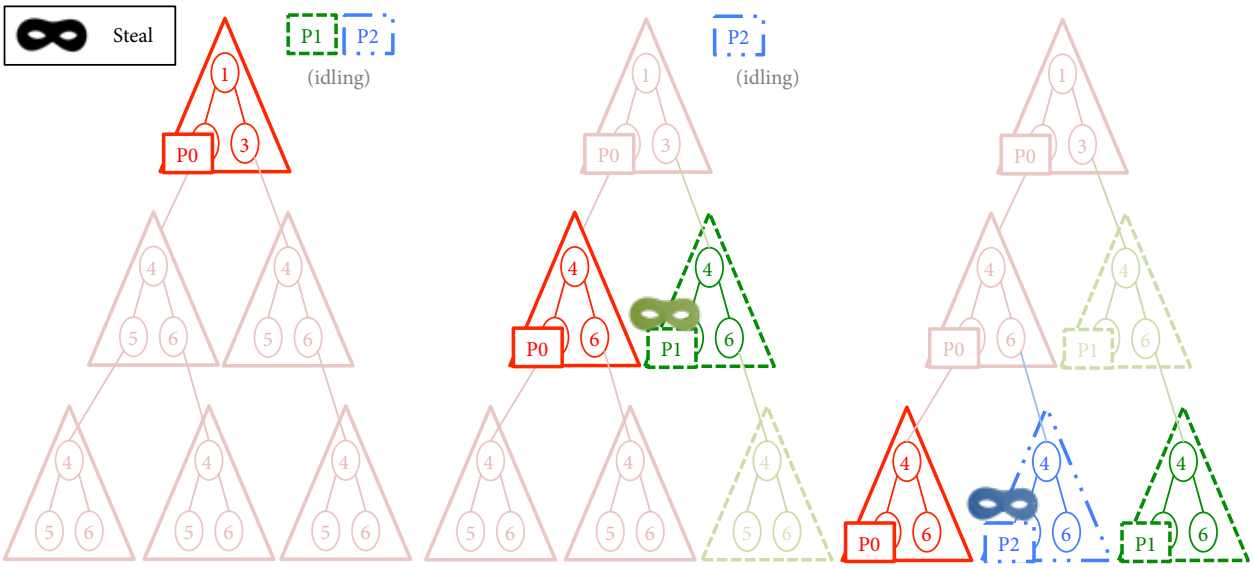


Figure 5.2: **Simulation of work stealing.** Top-down simulated tree traversal of a tiled tree by three processors in three steps.

- **Parallel scheduling.** Reasoning about individual nodes, such as for load balancing and synchronization, leads to high overheads. By scheduling tiles rather than nodes, we cut overheads. Nodes correspond to tasks in our system, so our approach is a form of *coarsening*. Furthermore, different synchronization strategies are possible for tiles, such as whether to use spin locks, so we autotune over the implementation options.

  We also support several scheduling options. First, we support third-party task schedulers, including Intel TBB [[CITE]], Cilk [[CITE]], and those of TesselationOS [[CITE]]. Second, we built our own that uses a variant of work-stealing threads pinned to processors. It includes options such as whether to use hyper threads or not, and as we saw low speedups when using multiple sockets, how many threads to use. Our autotuner picks between scheduler implementations.

Figure 5.1 depicts several of the data representation optimizations: packing, pointer compression, and a breadth-first layout.

Scheduling: Semi-static Work Stealing

We optimize our tree traversal task scheduler for low overheads, high temporal and spatial data locality, and load balancing. Webpages are relatively small and use many traversals, so we found that aggressively optimizing individual traversals to be an important implementation concern. Our approach is to combine static scheduling [[CITE]] with dynamic work stealing. We semi-statically schedule a traversal over the tree to as soon as it is available and

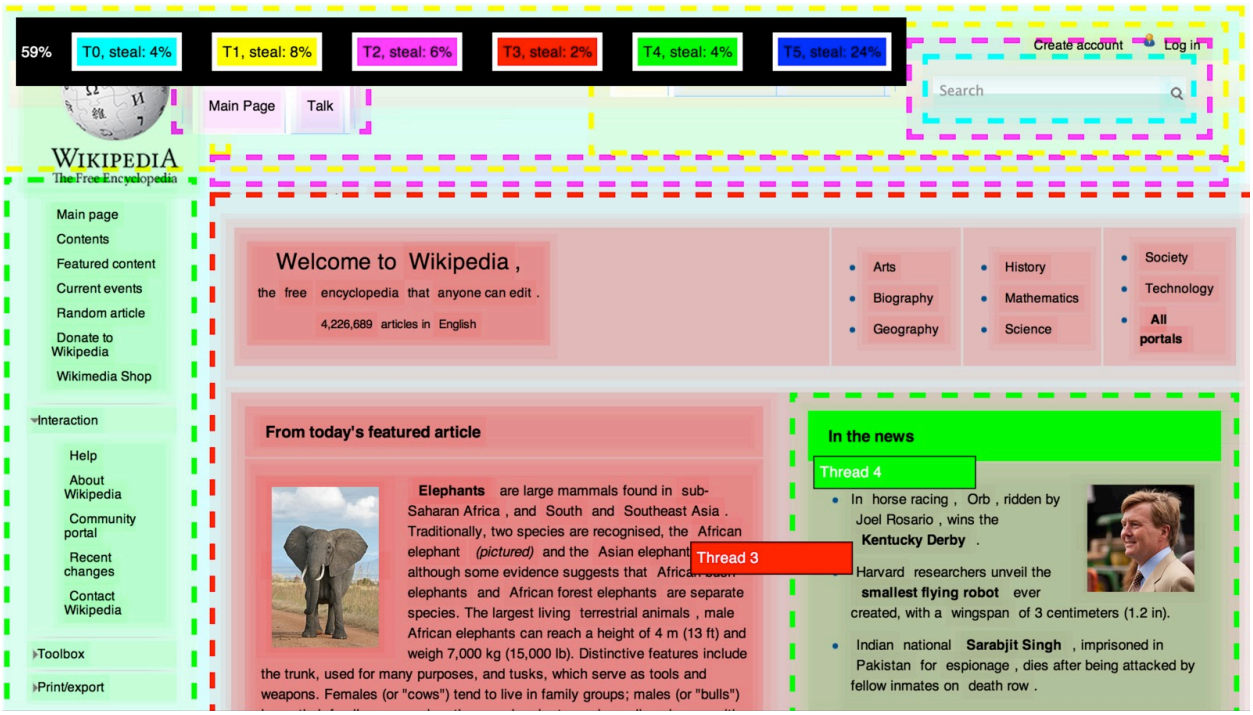


Figure 5.3: **Simulation of work stealing on Wikipedia.** Colors depict claiming processor and dotted boundaries indict subtree steals. Top-left boxes measure hit rate for individual processor.

reuse that schedule across traversals: this optimizes for temporal locality and low over-heads. We use work stealing as a heuristic for computing the first tree traversal for approximate load balancing. We did not see significant speedups with the base approaches on their own, but our combination led to 7X parallel speedups.

Our algorithm schedules the first traversal using work stealing [[CITE]]. Work stealing was introduced as a dynamic scheduling algorithm that provides load balancing and spatial locality. Figure 5.2 depicts a trace of three processors performing work stealing. Each processor operates on an internal task queue, and whenever a processor exhausts its internal queue, it will *steal* from another processor's queue. In the case of a top-down tree traversal, acting upon an internal queue corresponds to a depth-first traversal of a subtree, and stealing corresponds to transferring ownership of an untraversed subtree. We lower overheads on the first traversal in two ways: we perform task coarsening by scheduling tiles rather than individual nodes, and we simulate the work stealing in one thread on a localized copy of the tiling meta data. The colors of Figure 5.3 show how different processors claim different nodes of a webpage during a parallel traversal: the localization of colors demonstrates the spatial locality of work stealing. Likewise, figure demonstrates that there are relatively few scheduling overheads (steals are indicated by dotted borders).

Work stealing suffers from runtime overheads and lack of temporal locality. To estimate

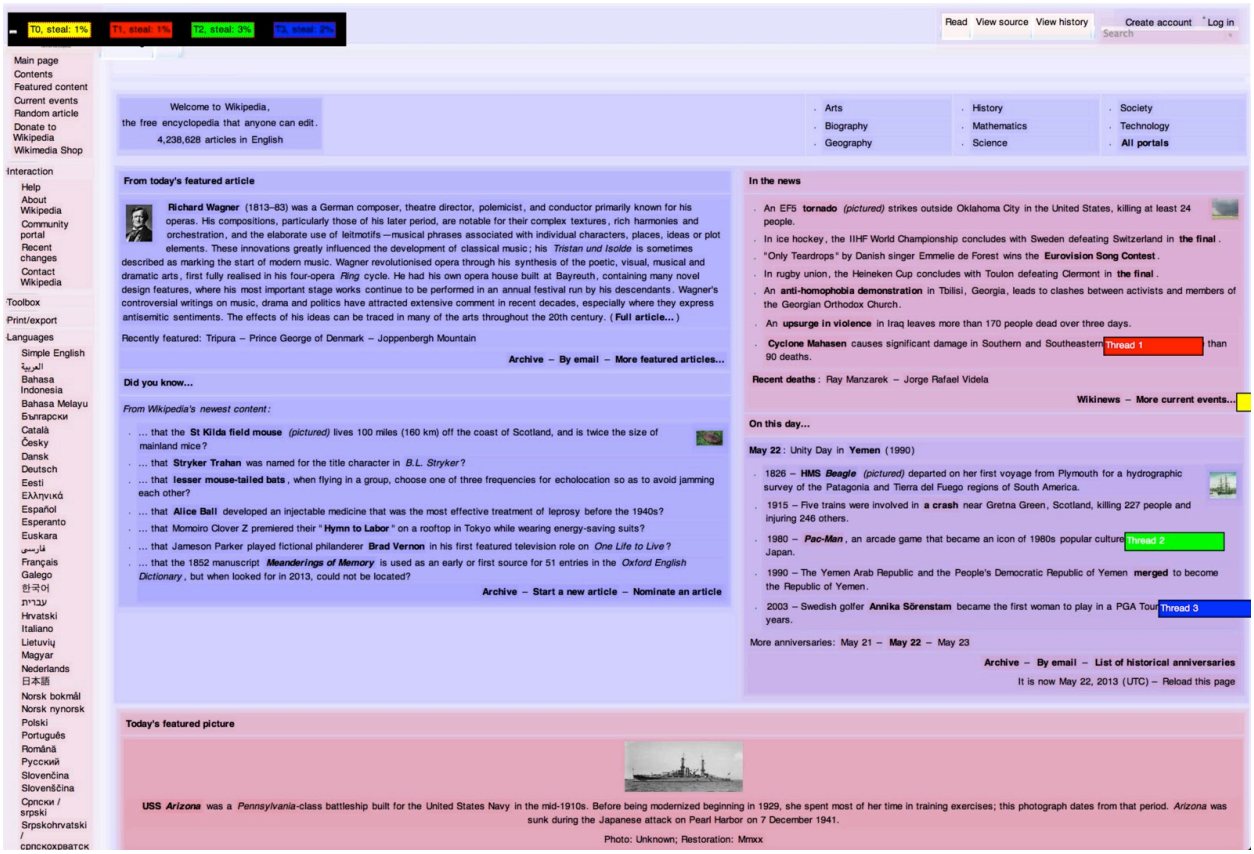


Figure 5.4: **Temporal cache misses for simulated work stealing over multiple traversals.** Simulation of 4 threads on Wikipedia. Blue shade represents a hit and red a miss. 67% of the nodes were misses. Top-left boxes measure hit rate for individual processor.

the overhead, we simulated work stealing for 6 processors on Wikipedia. Assuming uniform compute time per node, 5% of the nodes would trigger stealing. This cost is in addition to constant overhead to processing the internal per-processor task queues. The issue with temporal locality is that a node will be assigned to different processors across multiple traversals. Figure 5.4 shows which nodes must move across processors in a simulation 4 processors performing a sequence of two traversals. 67% of the nodes are red, indicating substantial movement. Both the steal rate and temporal miss rate worsen as the number of processors increase.

We use work stealing as a heuristic for semi-static scheduling so that the strengths of one address the weaknesses of the other. Semi-static scheduling precomputes the traversal order for each processor, which eliminates runtime overheads. Computing a load-balanced schedule can be quite expensive, however, because optimality is NP [[CITE]]. Instead, we use work stealing as a heuristic by running a simulation in which the cost of each tile is the

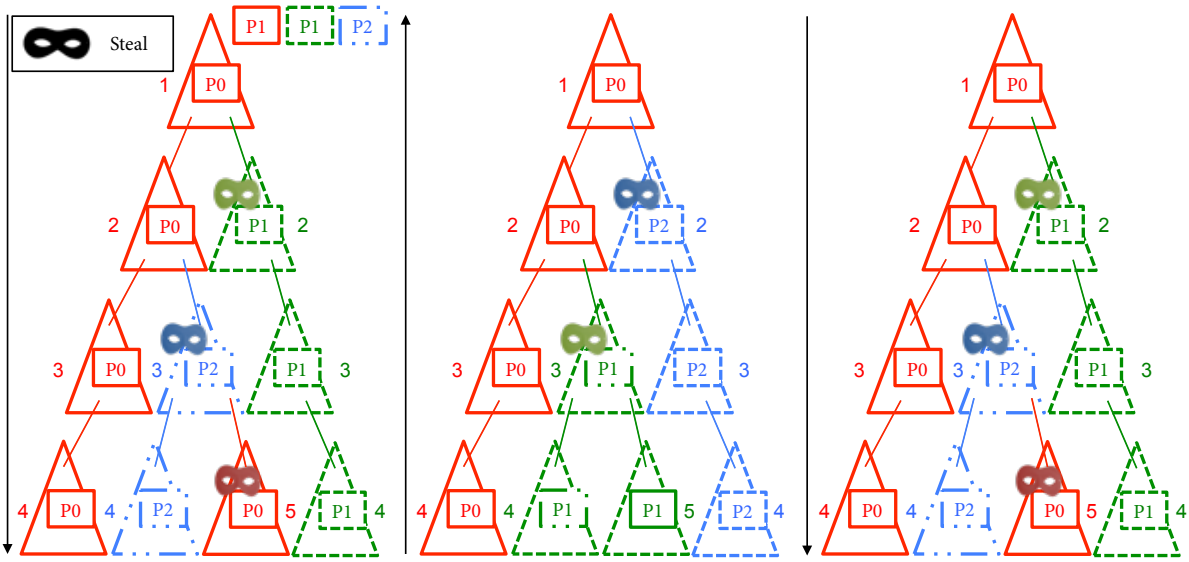


Figure 5.5: **Dynamic work stealing for three traversals.** Tiles are claimed by different processors in different traversals.

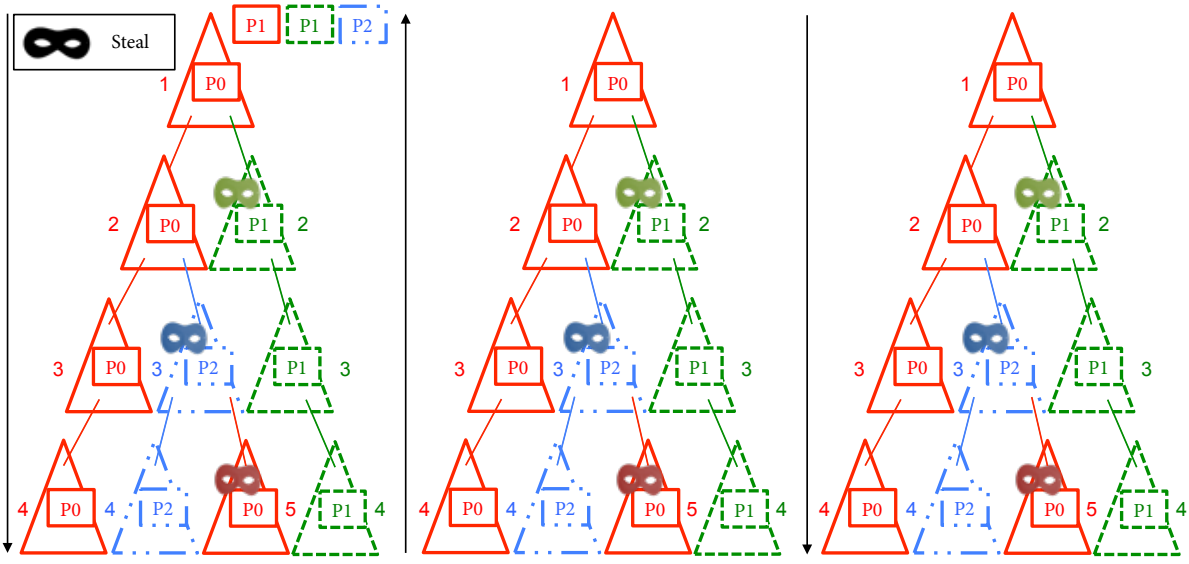


Figure 5.6: **Semi-static work stealing.** Dynamic schedule for first traversal is reused for subsequent ones.

number of nodes in it and penalizing simulated steals. The trace through the simulation for a top-down traversal is used as the schedule for top-down traversals, and the reverse for bottom-up. Computing the schedule is fast – a linear traversal over the tile meta data.

Our approach achieves low overheads, high temporal and spatial locality locality, and load balanced evaluation. Temporal locality is enforced by reusing the same schedule across the traversals, and semi-static scheduling with a fast heuristic provides low overheads. Our work stealing heuristic provides spatial locality and an approximate form of load balancing.

Evaluation

5.3 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.

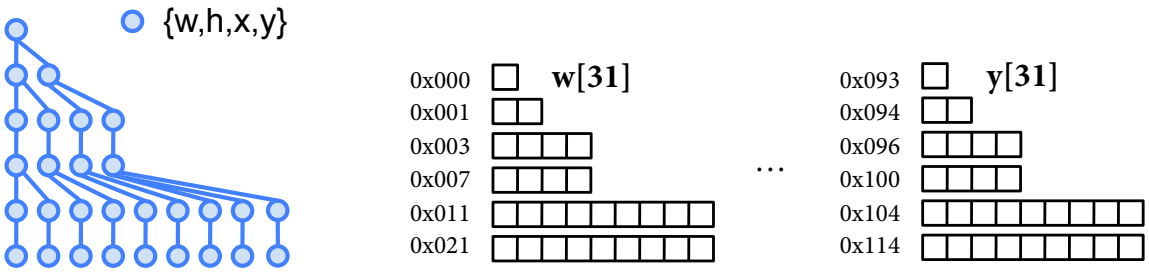
The naıve tree traversal schedule is to sequentially iterate one level of the tree at a time and traverse the nodes of a level in parallel. A parallel preorder traversal starts on the root node's level and then proceeds downwards, while a postorder traversal starts on the tree fringe and moves upwards (Figure 5.7 5.7a). 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 5.7c). 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 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.

```
1  void parPre(void (*visit)(Prod &), List<List<Prod>> &levels) {
2    for (List<Prod> level in levels)
3          parallel_for (Prod p in level)
4                visit(p)
5  }
6  void parPost(void (*visit)(Prod &), List<List<Prod>> &levels) {
7    for (Array<Prod> level in levels.reverse())
8          parallel_for (Prod p in level)
9                visit(p)
10 }
```

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



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

Figure 5.7: **SIMD tree traversal as level-synchronous breadth-first iteration with corresponding structure-split data representation.**

- **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.

5.4 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 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

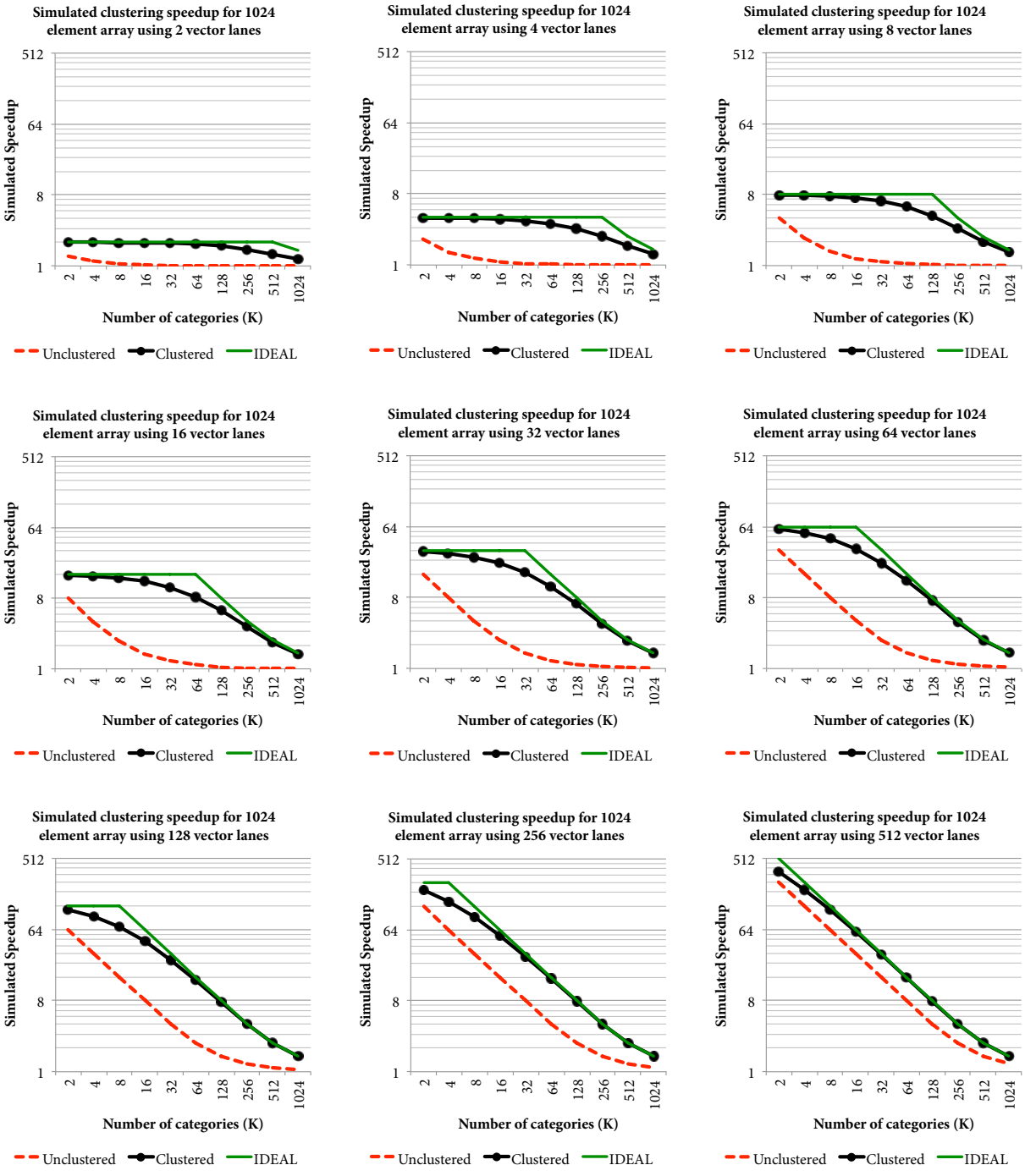


Figure 5.8: **blah**

```
1 void parPreClustered(void (*visit)(Prod &), List<List<Array<Prod>>> &levels) {
2   for (List<Prod> level in levels)
3         for (Array<Prod> cluster in level)
4               parallel_for (Prod p in cluster)
5                     visit(p)
6 }
```

Figure 5.9: **Clustered parallel preorder traversal.**

widths, while a VBox's is their maximum. The visit to a node (Figure **??**) 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 5.8 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 5.9 shows the clustered evaluation variant of the MIMD $parPre$ traversal of Figure **??**. 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 5.10 separates our transformation of $visit1$ (Figure **??**) into using the same exemplar for the dispatch and then loop unswitching.

```
1  Prod firstProd = cluster[0]
2  parallel_for (prod in Cluster) {
3    switch (firstProd.type) {
4      case S → HBOX: break;
5      case HBOX → ϵ:
6        HBOX.w = input();
7        HBOX.h = input();
8        break;
9      case HBOX → HBOX1 HBOX2:
10       HBOX0.w = HBOX1.w + HBOX2.w;
11       HBOX0.h = MAX(HBOX1.h, HBOX2.h);
12       break;
13   }
14 }
```

(a) **Clustered dispatch.**

```
1  Prod firstProd = cluster[0]
2    switch (firstProd.type) {
3      case S → HBOX: break;
4      case HBOX → ϵ:
5        parallel_for (prod in Cluster) {
6            HBOX.w = input();
7            HBOX.h = input();
8        }
9        break;
10     case HBOX → HBOX1 HBOX2:
11       parallel_for (prod in Cluster) {
12         HBOX0.w = HBOX1.w + HBOX2.w;
13         HBOX0.h = MAX(HBOX1.h, HBOX2.h);
14       }
15       break;
16   }
17 }
```

(b) **Unswitched dispatch**.

Figure 5.10: **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.

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.

5.5 Evaluation

MIMD Data Representation and Scheduling Optimizations

By statically exposing traversal structure (e.g., **parPre**) to our code generators, we observe sequential and parallel speedups. We separately evaluate the importance of the data representation optimizations from the scheduling ones on random 500-1000 node documents in the `hbox++` language. Finally, we examine the parallel benefit on webpages.

	Total speedup Cores				**Parallel speedup** Cores		
Configuration	1	2	4	8	2	4	8
TBB, server	1.2x	0.6x	0.6x	1.2x	0.5x	0.5x	1.0x
FTL, server	1.4x	2.4x	5.2x	9.3x	1.8x	3.8x	6.9x
FTL, laptop	1.4x	2.1x			1.6x		
FTL, mobile	1.3x	2.2x			1.7x		

Table 5.1: **Speedups and strong scaling across different backends (Back) and hardware**. Baseline is a sequential traversal with no data layout optimizations. FTL is our multicore tree traversal library. Left columns show total speedup (including data layout optimizations by our code generator) and right columns show just parallel speedup. Server = Opteron 2356, laptop = Intel Core i7, mobile = Atom 330.

		Parallel speedup Cores		
Backend	Input	2	4	8
TBB	Wikipedia	1.5x	1.6x	1.2x
TBB	xkcd Blog	1.5x	1.8x	1.2x
FTL	Wikipedia	1.6x	2.8x	3.2x
FTL	xkcd Blog	1.5x	2.3x	3.1x

Table 5.2: **Parallel CSS layout engine**. Run on a 2356 Opteron.

We first evaluate the perform of our task scheduler (FTL in Table 5.1). Our comparison point is Intel's TBB [**inteltbb**] dynamic task scheduler that performs work stealing [**cilk**], which was the most efficient third-party work stealing library that we tried. We included our data layout optimizations in all calculations because, without them, we saw no speedup. TBB causes slowdowns until achieving no cost (nor benefit) at 8 cores. Our insight is that it suffered from high overheads: switching to scheduling tiles by using our optimized data representation improved performance. Our semi-static working stealing scheduler, however, achieved a 6.9X speedup on 8 cores. We did not see significant further speedups for higher core counts, and hypothesize that it is due to the socket jump. We experimented with other schedulers, such as a simple for-loop over tiles near the fringe of the tree, but the achieved 2X speedup is much lower than the 6.9X of our semi-static work stealer.

Data representation was key to achieving parallel speedups. It achieved 1.2X-1.4X speedups for sequential processing (Table 5.1). However, on 4 cores, it improved performance from 2.8X without data representation optimizations to 5.2X when using them. The difference is 1.9X: our data representation optimizations both complement and improve scheduling optimizations. Without them, parallel performance was poor.

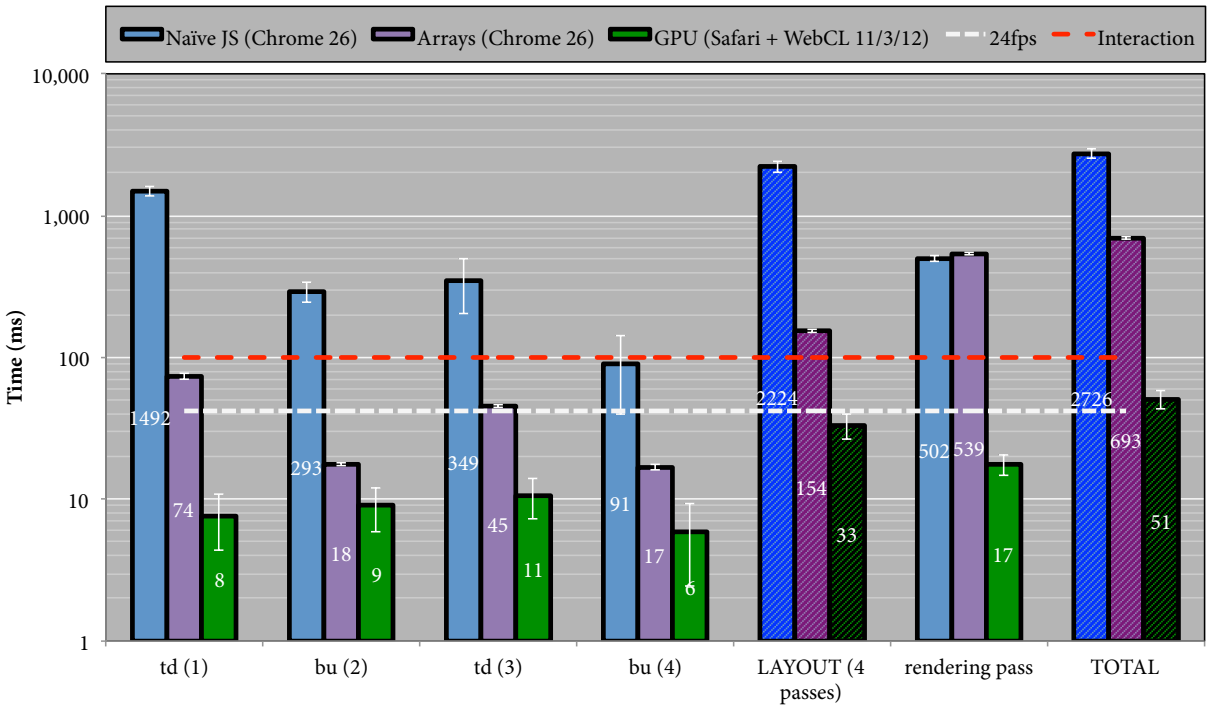


Figure 5.11: **Sequential and Parallel Benefits of Breadth-First Layout and Staged Allocation.** Allocation is merged into the 4th stage and buffer indexing and tessellation form the rendering pass. JavaScript variants use HTML5 canvas drawing primitives while WebCL does not include WebGL painting time (< 5ms). Thin vertical bars indicate standard deviation and horizontal bars show deadlines for animation and hand-eye interaction.

Table 5.2 shows the parallel speedup on running our 9 pass layout engine for two popular web pages that render faithfully with it: Wikipedia and the XKCD blog. Note that the benchmarks do *not* include sequential speedups. The best performance of TBB was a 1.8X speedup on 4 cores, and its speedup on 8 cores was 1.2X. In comparison, our scheduler achieved 2.8X on 2 cores and 3.2X on 8X. Our insight as to why we did not see further benefits is overheads. Across our benchmarks, we generally saw speedups when sequential traversals took longer than a certain amount, but because so many traversals are used for CSS, enough of them are small enough that we do not expect strong scaling. Our intuition is that either a full layout engine is complicated enough that the sequential cost of each traversal will be higher than in our prototype, or even more aggressive data representation optimizations should be performed. As is, we have demonstrated significant 3X+ speedups on real workloads from just the parallelization.

Baseline SIMD Speedups (GPU)

We evaluate the sequential and parallel performance benefits of our baseline breadth-first layout. For an animation to achieve 24fps, the time spent to process a frame should not exceed 42ms, and for eye-hand interactions, 100ms (10fps). We examine the case of a 5 pass treemap that supports live filtering over 100,000 data points. The first 3 passes are purely devoted to layout, the 4th pass includes layout computations and allocation requests, and the 5th pass propagates buffer indices and performs tessellation.

We compare 3 backends for our compiler: canonical JavaScript (a tree of nodes), JavaScript over our structure-split breadth-first tree layout (and with typed arrays [[CITE]]), and WebCL for the GPU. The first two variants invoke HTML5 canvas drawing primitives, while the last invokes WebGL (GPU) painting primitives over vertex buffers computed in the rendering pass. The time for WebGL painting calls are not shown, but they take less than 5ms. Each variant is repeated 15 times on a 4 core 2012 2.66GHz Intel Core i7 with 8 GB memory and a 1024 MB NVIDIA GeForce GT 650M graphics card.

We first examine the significant sequential benefits.The first 4 groups of columns in Figure 5.11 shows the average time spent on different layout passes and the 6th on the pass for buffer index computation and tessellation. Performing compiler optimizations enables a 14X sequential speedup on layout in the Chrome web browser. No speedup is observed in the rendering pass because the time is dominated by HTML5 canvas calls. We hypothesize part of the sequential benefit is related to our clustering optimization: all of the nodes in a level have the same type, so implicit optimizations such as branch prediction should perform better. Finally, we note that while sequential layout time is a magnitude too slow for real-time animation, our prototype is within 54ms for real-time interaction (ignoring rendering).

Parallel speedups are also significant. WebCL (GPU) evaluation of layout is 5X faster than sequential. The impact of compiling JavaScript vs. C (WebCL) on the benchmark is unclear: JavaScript is generally a magnitude slower than native code, except the runtime WebCL compiler is not running at high optimization levels. The benefits for parallel computation of the buffer indices and tessellation is much more clear: the speedup is 31X.

To better understand the benefit of parallelization, we compared running the layout traversals using multicore vs. GPU acceleration (Figure 5.12) for an early prototype of the layout traversals. Both use breadth-first traversals compiled with OpenCL, except differ on the hardware target. We see that a server-grade multiprocessor (32-core AMD Opteron 2356) can outperform a laptop GPU, but the comparison is unfair in terms of power consumption. TODO compare power ratings.

Ultimately, when the sequential and parallel optimizations are combined, we see an end-to-end speedup of 54X. It is high enough such that it enables real-time animation for our data set, not just real-time user interaction.

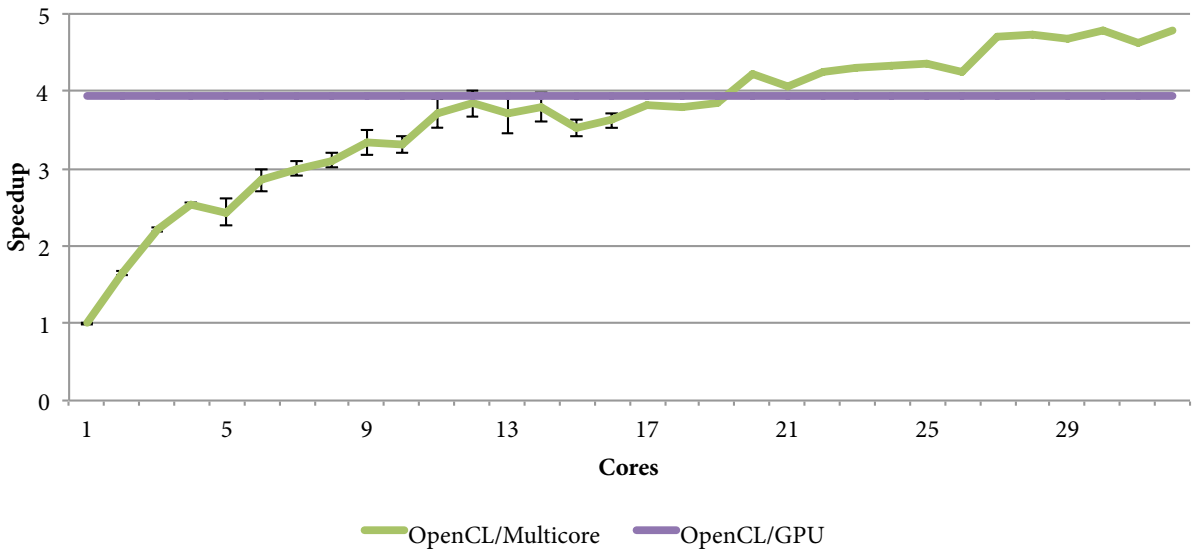


Figure 5.12: **Multicore vs. GPU Acceleration of Layout.** Benchmark on an early version of the treemap visualization and does not include rendering pass.

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 5.8 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

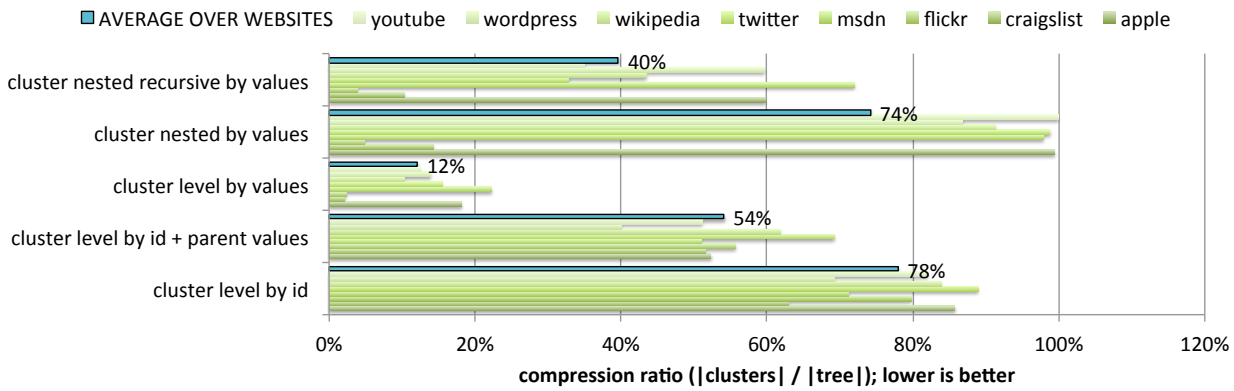


Figure 5.13: **Compression ratio for different CSS clusterings.** Bars depict compression ratio (number of clusters over number of nodes). Recursive clustering is for the reduce pattern, level-only for the map pattern. ID is an identifier set by the C3 browser for nodes sharing the same style parse information while value is by clustering on actual style field values.

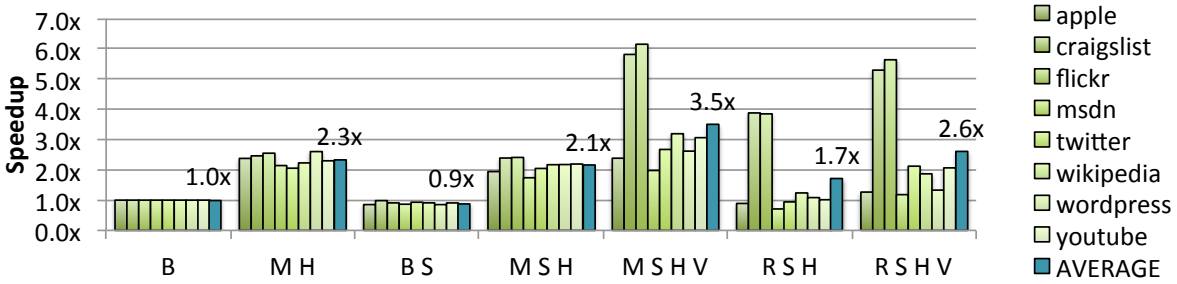
multiple line graphs puts the root node on the first level, the axis for each line graph on the second level, and all of the actual line segments on the third level.

- **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

[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 5.14: **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.

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 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 5.14). However, clustering each level and hoisting computations shared throughout a cluster led to a 2.1X sequential benefit (M S H).

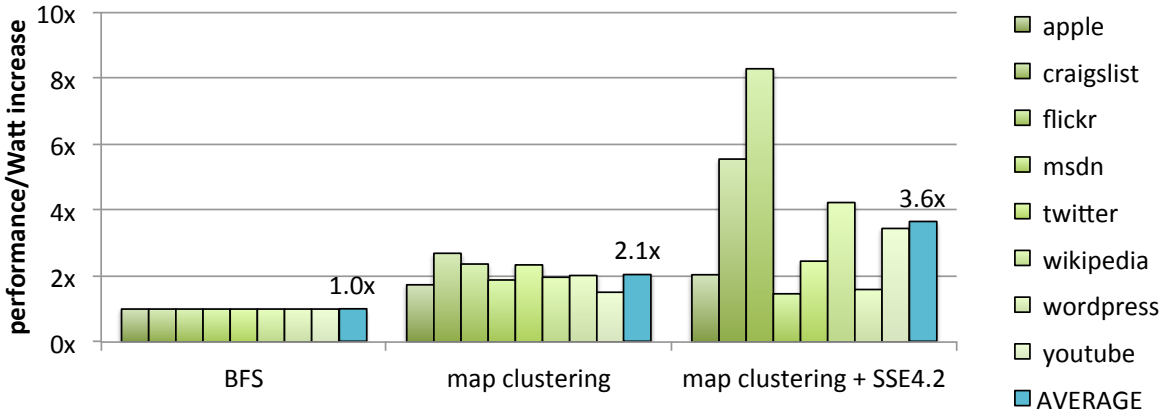


Figure 5.15: **Performance/Watt increase for clustered webpage layout.**

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.

In Figure 5.15, we show a 2.1X improvement in power efficiency for clustered sequential evaluation, which matches the 2.1X sequential speedup of Figure 5.14. 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.

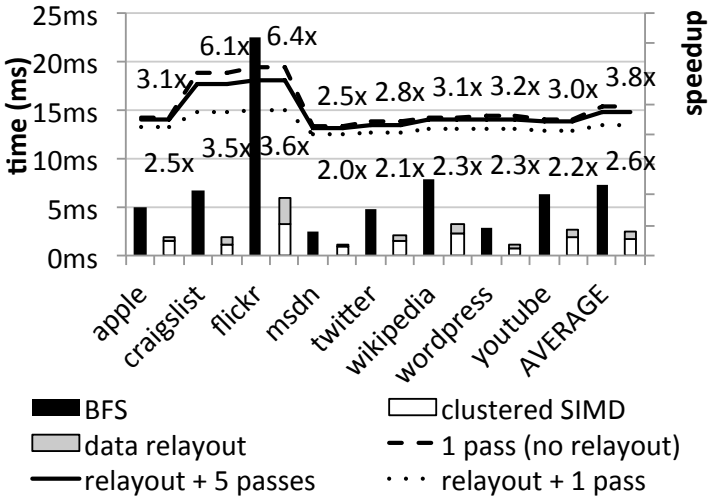


Figure 5.16: **Impact of data relayout time on total CSS speedup.** Bars depict layout pass times. Speedup lines show the impact of including clustering preprocessing time.

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 5.16 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 5.16 show there is little difference when we include the preprocessing overhead in the speedup calculation.

5.6 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