# Parallel Schedule Synthesis for Attribute Grammars

Leo A. Meyerovich, Matthew E. Torok, Eric Atkinson, Rastislav Bodík

University of California, Berkeley *

{lmeyerov,mtorok,ericatkinson,bodik}@eecs.berkeley.edu

## Abstract

We examine how to synthesize a parallel schedule of structured traversals over trees. In our system, programs are declaratively specified as attribute grammars. Our synthesizer automatically, correctly, and quickly schedules the attribute grammar as a composition of parallel tree traversals. Our downstream compiler optimizes for GPUs and multicore CPUs.

We provide support for designing efficient schedules. First, we introduce a declarative language of schedules where programmers may constrain any part of the schedule and the synthesizer will *complete* and *autotune* the rest. Furthermore, the synthesizer answers debugging queries about how schedules may be completed.

We evaluate our approach with two case studies. First, we created the first parallel schedule for a large fragment of CSS and report a 3X multicore speedup. Second, we created an interactive GPU-accelerated animation of over 100,000 nodes.

***Categories and Subject Descriptors*** I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program Synthesis; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

***Keywords*** CSS, layout, sketching, attribute grammars, scheduling

## 1. Introduction

Programmers struggle to map applications into parallel algorithms. We examine attribute grammars, which are a declarative formalism for defining tree processors such as document layout engines. A grammar is high-level because it does not specify a tree traversal order that computes all node attributes. Algorithm designers optimize parallel tree traversals, so our approach is to stage the problem by first finding a schedule of traversals for a grammar.

We present a synthesizer that automatically schedules an attribute grammar as a tuned choice of tree traversals. For example, if our synthesizer schedules a grammar as a sequence of parallel preorder tree traversals, our more traditional GPU compiler can then implement them as level-synchronous breadth-first tree traversals. As another example, we present a case study of synthesizing

* Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung.

*PPoPP'13*, February 23–27, 2013, Shenzhen, China.
Copyright © 2013 ACM 978-1-4503-1922-5/13/02...$15.00

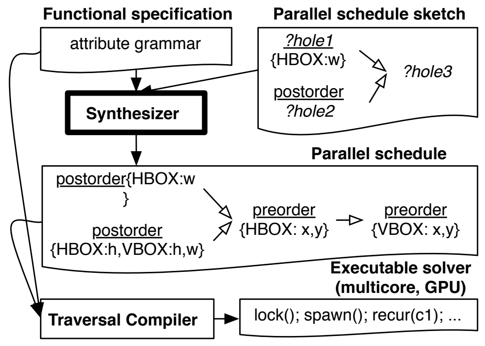


**Figure 1: Synthesizer input/output:** The synthesizer completes a schedule sketch and gives it to a traditional parallel compiler.

a schedule for a multicore CSS [3] webpage layout engine. CSS is a long-standing sequential bottleneck in web browsers that consumes 15-22% of the CPU time [9, 21]. We found the need to guide the choice of schedule, so we extend grammars with a language of schedules where programmers may specify parts of the schedule and let the synthesizer fill in the rest.

Our synthesizer explores the space of scheduling decisions:

- **Decomposing a sequential 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. Our synthesizer therefore explores multiple non-obvious alternatives.
- **Composition of 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. Our synthesizer selects by autotuning.
- **Distribution of computations across traversals.** Even for a fixed schedule, i.e., a composition of traversals, node computations might commute across traversals. Redistributing them may improve memory use and avoid sequential bottlenecks.

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.

We present three techniques for automatically synthesizing a parallel schedule for an attribute grammar (Figure 1):

***1: A scheduling language of parallel tree traversals that is explicit, orthogonal, and safe.*** A program can fully specify a *schedule* to simplify code generation. It identifies the available parallelism (composition of traversal types). The schedule is still high-level: it is specified alongside the functional specification and compilers handle the actual translation into lower-level code. We adapt attribute grammar dependence analysis [13] to statically verify that a schedule respects dependencies in the attribute grammar.

***2: Schedule sketching for automatic parallelization.*** Fully automatic scheduling obstructs programmer guidance. We extend our scheduling language with a *sketching* [25] construct for partial specification of schedules. A *hole* is a symbolic variable that can be put in place of any term of an otherwise fully specified schedule. As an extreme example, the entire schedule may be specified as a hole. Our synthesizer fills in any holes to achieve a fully specified schedule. It finds a correct completion, and if there are multiple functionally correct ones, autotunes for the fastest. The same sketching language acts as a query language for parallelism debugging. Our synthesizer outputs sketch completions, and if the sketch and functional specification are mutually inconsistent, it shows the earliest stage in which the sketched schedule is incompletable.

***3: Fast and extensible schedule synthesis.*** We struggled to implement a synthesizer that is fast and can be extended with new traversal types. Ours supports inputs and outputs beyond those of current grammar compilers, such as schedule sketches as input constraints, returning multiple schedules as outputs, and parameterization by multiple traversal types and composition operators.

Synthesis is a simple search: A) *enumerate* increasingly complete schedules and B) invoke individual traversal verifiers to *check* every partial schedule. We optimize both steps so synthesis is $O(n^3)$ in the number of attributes. To synthesize exponential-time extensions such as nesting schedules, we use incrementalization, sketching, and greedy heuristics.

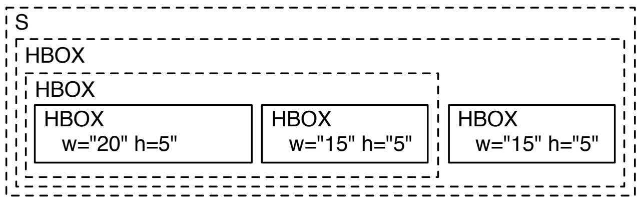
In summary, we present a synthesizer that automatically schedules an attribute grammar as a tuned choice of tree traversals. First, we introduce a language of *structured* traversal schedules that can be sketched and verified (Section 2). Second, we describe our extensible $O(n^3)$ algorithm for synthesizing a correct schedule (Section 3). Finally, we use the synthesizer for schedule autotuning, parallelism debugging, and new primitives (Section 4).

We evaluate our approach with two case studies (Section 5). First, we synthesized a parallel schedule for a fragment of the CSS webpage layout language. CSS has long challenged parallelization [21]. We report multicore speedups of 3X. Second, we implemented a GPU-accelerated interactive visualization of $10^5$ nodes.

## 2. Parallel Programming with Synthesis

Our system can be understood in terms of our case study of synthesizing a webpage layout engine. Accordingly, we present a running example of synthesizing the schedule for a simple layout language.

Its architecture is in Figure 1. A webpage is a tree with style constraints over attributes on each node, which the layout engine solves during tree traversals. Given the attribute grammar specifying a layout language, our synthesizer finds a parallel schedule. Next, our compiler that optimizes for different traversal patterns reads the schedule and outputs a browser layout engine that executes the traversals. In summary, our system uses several functions:



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

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

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

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

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

*AG* $\rightarrow$ (*Prod* **{** *Stmnt*? **}**)*
*Prod* $\rightarrow$ *V* $\rightarrow$ *V**
*Stmnt* $\rightarrow$ *Attrib* = *id*(*Attrib**) | *Attrib* = *n* | *Stmnt* ; *Stmnt*
*Attrib* $\rightarrow$ *id.id*

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

**Figure 2:** (a) input tree (b) attribute grammar specifying the layout language (c) specification language of attribute grammars

$$\text{Synthesizer: } AG \rightarrow Sched \quad (1)$$
$$\text{Compiler: } AG \times Sched \rightarrow LayoutEngine \quad (2)$$
$$\text{LayoutEngine: } ConstraintTree \rightarrow ConcreteTree \quad (3)$$

This section describes specifying a layout language as an attribute grammar and the synthesized parallel schedule. We found implementing more complicated layout languages benefits from support in controlling and reasoning about the schedule. Thus, we conclude this section by extending the input language to support *sketching* (partial specification) of the schedule. A sketch is a full schedule, except any term (e.g., a choice of traversal type) can be left as a hole (symbolic variable) that the synthesizer will fill in:

$$\text{Synthesizer}_{sketch}\text{: } AG \times Sched_{sketch} \rightarrow Sched \quad (4)$$

### 2.1 Attribute Grammars

Consider solving the tree of horizontal boxes shown in Figure 2 (a). As input, a webpage author provides a constraint tree. Only some node attribute values are provided: the widths and heights of leaf nodes. The meaning of a horizontal layout is that, as is also depicted, the boxes will be placed side-by-side. The layout engine must solve for all remaining x, y, width, and height attributes.

The layout language of horizontal boxes, `H-AG` (Figure 2 (b)) can be declaratively specified as an attribute grammar [15, 21, 24]. First, the specification defines the set of well-formed input trees as the derivations of a context-free grammar. 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

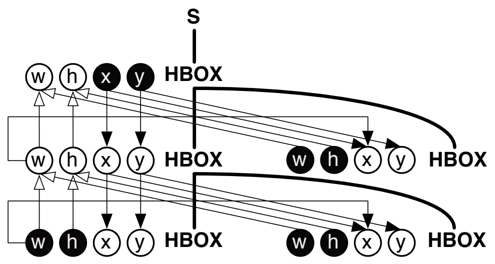


**Figure 3: Data dependencies.** Shown for constraint tree in Figure 2 (a). Circles denote attributes, with black circles being input() sources. Thin lines show data dependencies and thick lines show production derivations.

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 nullary function call $input_w()$:

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

$HBOX_0 \rightarrow HBOX_1\ HBOX_2$ /* binary node */
{ . . . $\mathrm{HBOX}_0$.w = $\mathrm{HBOX}_1$.w + $\mathrm{HBOX}_2$.w }

Note that the evaluation order is not specified. For example, while the above statements will be executed within different tree traversals, the mapping is not specified. Likewise, an executable implementation may need to reorder the statements within a traversal. Whatever evaluation order is used to solve for the attribute values, the statements are constraints that must hold over them. Attribute grammars can therefore be thought of as a single assignment language where attributes are dataflow variables.

The language of attribute grammars is defined in Figure 2 (c). Our example assumes the following encoding. Semantic functions are uninterpreted, so, for example, the addition of widths can be rewritten as "$\mathrm{HBOX}_0$.w = f($\mathrm{HBOX}_1$.w, $\mathrm{HBOX}_2$.w)". Likewise, constant values are equivalent to nullary function calls. To specify grammars more complicated than H-AG , we provide extensions (Section 4) whose scheduling reduces to attribute grammars.

## 2.2 Language of Schedules

Given an attribute grammar, our synthesizer statically finds a schedule of tree traversals that will, for any tree described by the grammar, solve all of its attributes. For example, the width and height attributes of any H-AG tree can be solved in an initial *postorder* tree traversal, after which the x and y attributes can be computed with a *preorder* tree traversal. This two-pass schedule respects all data dependencies possible in an input tree. For example, it respects the data dependencies for Figure 2 (a), which are shown in Figure 3. Finally, both traversals exhibit structured parallelism that a compiler can exploit: parallel preorder allows a top-down wavefront, and postorder allows bottom-up.

Our synthesizer targets a language of traversals. A schedule for H-AG that exercises different types of traversals is in Figure 4 (a). It declares what simple tree traversals to use (parallel preorder and parallel postorder); how to combine them (here, serially); and what attributes to compute when a production is visited in each of these traversals. This section also describes parallel and nested combination as well as sequential recursive traversals. Our synthesizer is designed to support adding even more variants (Section 3).

A schedule is fed to simple compilers that produce evaluation code. An evaluator has two parts, as Figure 4 (c) shows for H-AG :

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

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

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

**(b) Naïve traversal implementations** with Cilk's [2] **spawn** and **join**.

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

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

*Sched* $\rightarrow$ *Sched* ; *Sched* | *Sched* || *Sched* | *Trav*

*Trav* $\rightarrow$ *TravAtomic Visit**{(*TravAtomic* $\mapsto$ *Visit**)*}?

*TravAtomic* $\rightarrow$ **parPre** | **parPost** | **recursive**

*Visit* $\rightarrow$ *Prod* { *Step** }

*Step* $\rightarrow$ *Attrib* | **recur** *V*

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

**Figure 4: Scheduled and compiled layout engine for H-AG .**

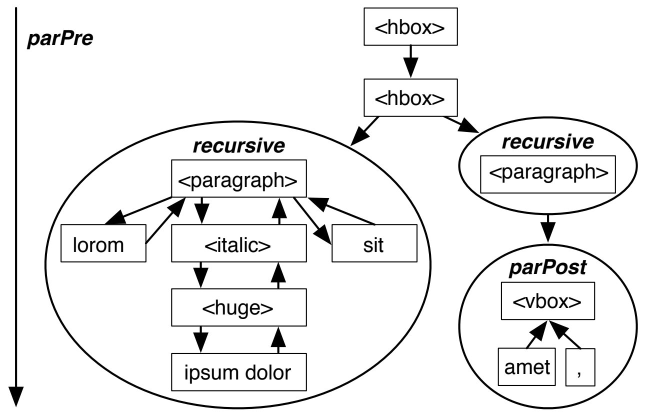


**Figure 5: Nested traversal for line breaking**. The two paragraph are traversed in parallel as part of a preorder traversal and a sequential recursive traversal is used for words within a paragraph.

1. **The traversals to execute**. Line 25 shows the sequence of two traversals for `H-AG`. An individual traversal can be parallel, as shown for the naïve implementations for preorder and postorder traversals in Figure 4 (b). Our compilers use the traversal structure to safely apply further optimizations (Section 5.2).
2. **Statements to execute within a traversal.** Lines 17-22 show the statements run for intermediate nodes in the second traversal. Every attribute in the schedule corresponds to a unique statement's left-hand side attribute. The compiler automatically infers the order of statements by topologically sorting them according to their dependency graph (Section 3.2).

Multiple schedules may be correct. For example, the initial **parPost** traversal for `H-AG` can instead be scheduled as two concurrent **parPost** traversals:

```
(     parPost
        $HBOX_0 \rightarrow HBOX_1\ HBOX_2$  { $HBOX_0$.w }
        HBOX $\rightarrow \epsilon$  { HBOX.w }
    ||
      parPost
        $HBOX_0 \rightarrow HBOX_1\ HBOX_2$  { $HBOX_0$.h }
        HBOX $\rightarrow \epsilon$  { HBOX.h })
; parPre ... /* same as before */
```

Prior attribute grammar compilers would partition attributes into independent sets [13], but not as part of a greater schedule.

We provide two traversal types in addition to **parPre** and **parPost**. The first is a sequential **recursive** traversal. We use it, for example, in our case study of document layout. Consider inserting line breaks into the following stylized paragraph of XML strings:

```
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 5 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.

Our second traversal type is a *nested* traversal. 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-nodes that are evaluated in full (with their particular traversal type) when the outer traversal encounters them.

To motivate the need for the nested traversal type, 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 paralle, while each paragraph forms an inner region that is handled with the sequential recursive traversal. Figure 5 shows how parallel evaluation may be used to compute across different **recursive** paragraphs and within individual **parPost** regions for this example.

To partition a tree into regions, each grammar production (and thus each node of the tree) is mapped 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:

```
parPre
  P $\rightarrow$ W { W.relativeX }
  { recursive $\mapsto$
      $W_0 \rightarrow W_1\ W_2$ {
        $W_1$.relativeX  recur $W_1$
        $W_2$.relativeX  recur $W_2$ } }
```

Overall, we see that schedules are explicit, orthogonal, and safe. They are explicit in that a custom code generator can be invoked without further high-level analysis. They are orthogonal in that they sit alongside the grammar: changes to one are often made independent of the other. Finally, they are safe. Traditional attribute grammar analyses can be used to check that a schedule does not violate a dependence in the attribute grammar (Section 3). If there is a bug, it is in the functional specification as an attribute grammar or somewhere in the implementation of the compiler toolchain.

### 2.3 Sketching

Specifying a full schedule is difficult. There are many attributes to schedule within a traversal, and often times, it is unclear whether a schedule is possible. We introduce a sketching language where programmers can specify parts of the schedule they care about and leave a hole, *?hole*, anywhere else. The synthesizer will fill in the hole, even if the hole is the entire schedule.

We can concisely specify the preceding schedules as follows:

$$\textbf{parPost}\ ?hole_1\ ;\ \textbf{parPre}\ ?hole_2 \tag{5}$$

$$(\textbf{parPost}\ ?hole_3\ ;\ \textbf{parPost}\ ?hole_4)\ ;\ ?hole_5 \tag{6}$$

$$?hole_6 \tag{7}$$

In the first example, the synthesizer finds that the width and height attributes can be computed in $?hole_1$ and the remaining in $?hole_2$. The second example can be completed in several ways. Most prominently, $?hole_5$ may be a **parPost** or a sequential **recursive** traversal. By default, our synthesizer picks the first parallel traversal it finds, and our autotuner (Section 4.1) can optimize the choice.

Sketching has various uses. The third variant, $?hole_6$, enables automatic parallelization, which helps with prototyping. As an attribute grammar grows and is shared by programmers, sketches enable static checks that program edits do not break the parallelization scheme. Finally, sketches speed up synthesis time (Section 3.1).

## 3. Schedule Synthesis Algorithm

Our synthesizer takes an attribute grammar and a sketch as input, and outputs a set of schedules. It is designed to support multiple traversal types, multiple solutions, and rich attribute grammar and schedule sketching languages. Our initial implementation used the dependency analysis of Kastens [15], but it was too inflexible. Our new algorithm is designed for modularity and speed:

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

***Simple enumerate-and-check*** The algorithm enumerates schedules and checks which are correct. Checking is for individual traversal types and for traversal compositors, and checkers are written independently of one another. Enumeration is simply syntactic. Adding a new traversal type involves adding a checker and syntax.

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

We now overview our high-level algorithm. After, we describe how to check the correctness of an overall schedule and individual traversal, and then analyze the correctness of our optimizations.

## 3.1 The Algorithm

We split discussion of optimizations between finding one schedule and finding many. Figure 6 demonstrates an algorithm trace for enumerating schedules of `H-AG` . Figure 9 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 "**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 $\mathsf{check}_\beta$ 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 can be performed at most $A$ times before reaching the empty set. Checking one refinement invokes the $O(A)$ topological sort. Put together, finding 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 $O(A^3)$.

Features such as nesting regions and enumerating all schedules are exponential, which we address with three further optimizations:

1. **Backtracking.** To emit multiple schedules, prefix expansion is modified to backtrack. After a schedule is 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 an alternative suffix in line 9.
2. **Interleaved sketch unification.** Sketching prunes the search. For example, "**parPost** *?hole* || *?hole*" 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 checking, which is $O(A)$. 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 is still only $O(A^3)$.
3. **Greedy 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.

In summary, synthesizing one schedule is $O(A^3)$, while emitting all of them is exponential. Finally, constructs such as nested traversals are still efficient when guided by sketches.

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

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

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

$$\frac{\begin{array}{c} P = \cup Prod_i \quad Steps = \cup Step_j \\ B = \bigcup_i \texttt{reachable}_\beta(Prod_i, P, A, Steps, C) \end{array}}{\{A, C\}\ \ \beta\ (Prod_i\ \{\ Step_j \ast\ \})\ast\ \ \{A \cup B\}}\ \text{(check}_\beta\text{)}$$

$$\frac{\{A\}\,p\,\{B\} \quad unify(sketch, p)}{\{A\}\,p \wedge sketch\,\{B\}}\ \text{(sketch)}$$

**Figure 7:** Correctness axioms for checking a schedule

```
1 alwaysCommunicate_parPre(β, B, M) =
2     $\{a_{W,W\to X} \mid (W\to X\ B_\beta) \in M[\beta]$
3         $\wedge \bigwedge_{(V\to W\ B_\gamma)\in M[\gamma\neq\beta]} a_{W,V\to W} \in B \cup A\}$
```

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

```
1  set reachable_parPre($W\to X$, $P$, $A$, $B$, $C$):
2    reach :=
3        $\{a_{*,W\to X} \mid a_{*,W\to X} \in A\}$
4      $\cup\, (C \cap \{a_{W,W\to X} \mid \bigwedge_{V\to W\in P} W.a_{V\to W} \in B\})$
5      $\cup\, (C \cap \{a_{X,W\to X} \mid \neg\exists X\to Y \in P\})$
6    while true:
7      progress := $\{a_{*,W\to X} \mid a_{*,W\to X} = f(b_0,\ldots,b_n) \in$ F
                  $\wedge\, a_{*,W\to X} \in B\ \wedge\ \bigwedge b_i \in$ reach}
8      reach := reach $\cup$ progress
9      if progress = $\emptyset$:
10       break
11   return reach
```

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

**Figure 8:** Inter- and intra-region checkers for **parPre**.

## 3.2 Correctness Checking Axioms

Correctness axioms for checking an entire schedule are in Figure 7. The judgements recursively check a composition of traversals until reaching the traversal-specific checks of Figure 8. This procedure is inefficient and monolithic; the next subsection will why our optimizations correctly interleave the checks presented here.

Variables $p$ and $q$ denote schedules (<Sched>), $A$ and $B$ are sets of attributes, and $\alpha$ and $\beta$ 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 composition and traversal rules 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*** Rule nest$_\alpha$ checks outer traversal type $\alpha$ over regions where each one may have its own traversal type $\gamma$. 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 $\gamma$), the rule calls alwaysCommunicate$_{\text{parPre}}$ to find the set $C_\gamma$ of attributes that are externally set before the region is traversed. Rule nest$_\alpha$ calls checks for every region under the assumption that $C_\gamma$ is already solved.

The first line of rule nest$_\alpha$ means that, for any outer traversal $\alpha$, 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*** The schedule for a traversal of type $\beta$ over a region is correct if every production visit schedule is correct (rule check$_\beta$). A production visit schedule $Prod_i\ \{\ Step_j \ast\ \}$ 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.

***Production visit*** Figure 8 (b) 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. The checks for the intra-region case and the alwaysCommunicate inter-region case are in lines 4-5 of nest$_\alpha$.

***Sketches*** Rule sketch separates checking the correctness of a schedule from whether a sketch matches it. First, a schedule must be correct irrespective of the accompanying sketch. Second, the schedule must syntactically match the schedule ($unify$). Later, we provide semantically constrained sketches that are checked by Prolog's more general unifier (Section 4.2).

## 3.3 Correctness of the optimizations

The optimization are *sound* and *complete* with respect to the axioms in Figure 7. Soundness means there is a derivation tree for a synthesized schedule, and completeness mean the optimizations do not preclude sound schedules. Most of our optimizations prune schedules from consideration by moving checks earlier, which is sound, so we only manually analyze completeness here.

***Prefix expansion*** Prefix expansion prunes schedules $p \otimes q$ if $p$ does not check. Completeness has two important cases. First, pruning a failing prefix $p$ does not prune sound schedules. Any expansion $p \otimes q$ would have been rejected because composition operator check would fail. Second, to synthesize only one schedule, only one increasingly long prefix $p$ needs to be expanded. Assume some alternative prefix $q$ succeeds. A sound completion to $p$ would be $q$ modified to not include attributes already solved by $p$. If a sound schedule exists, prefix expansion will return one.

***Incremental checking*** Incremental checking is sound and complete because it is the memoization of checking $p$ for all completions $p \otimes q$.

***Iterative refinement*** Refinement is complete because it only removes attributes from consideration that cannot be scheduled. Consider refining unreachable attribute $a$ found by rule $\mathsf{check}_\beta$:

$$a \in \ B - \bigcup_i \texttt{reachable}_\beta(Prod_i, P, A, Steps, C)$$

If an alternative traversal computes a subset of $Steps$ and $a$, the assumption of what is reachable before $a$ is weakened. Attribute $a$ will again be unreachable, and the schedule will fail. Refinement prunes such schedules, and therefore does not affect completeness.

***Interleaved unification with sketches*** Interleaving is complete because any rejected schedule would have failed for the corresponding unification check.

***Greedy heuristic*** By design, the greedy heuristic is not complete. Instead, classical attribute grammar languages [15] use greediness to guarantee that a node is visited a minimum number of times. We support traversal types of varying strengths, so this property is not immediate. For example, "**recursive** ; **parPre**" can often (but not always) be replaced by "**parPre** ; **parPost**; **parPre**", which is longer but more parallel. We instead guarantee that if there is a shorter schedule, it uses different traversal types.

```
1  def synthFast(sketch):
2    yield synth($\emptyset$, Attributes, sketch)

4  def synth(prev, rest, sketch):
5    choose  $\otimes \in$ { ";", "||" }
6    if $\otimes$ = ";":
7      choose $\alpha \in$ { "parPre", "parPost", ... }
8      $A$ := iterativeRefine($\alpha$, prev, rest)
9      if $A$ = rest:
10       unify(sketch, $\alpha A$)
11       yield $\alpha A$
12     else if $A = \emptyset$:
13       backtrack
14     else:
15       unify(sketch, $\alpha A$ ; $rhs_1$)
16       yield $\alpha A$ ; synth(prev $\cup A$, rest $- A$, $rhs_1$)
17   else:
18     unify(sketch, $lhs_2$ || $rhs_2$)
19     choose $A \subset$ rest
20     $p$ := synth(prev, $A$, $lhs_2$)
21     $q$ := synth(prev, rest   $A$, $rhs_2$)
22     yield $p$ || $q$

24 def iterativeRefine($\alpha$, prev, rest):
25   overapproxA = rest
26   do:
27     X = check$_\alpha$(prev, overapproxA)
28     overapproxA = overapproxA $-$ X
29   while X $\neq \emptyset$
30   yield overapproxA
31   if nonGreedy:
32     choose overapproxA$'$ $\subset$ overapproxA
33     yield iterativeRefine($\alpha$, prev, overapproxA$'$)
```

**Figure 9: 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.

## 4. Extensions

We outline three extensions that use the attribute grammar synthesizer: a schedule autotuner, a parallelism debugger, and grammar extensions for classes, loops, and schedule constraints. These extensions are important because they influenced the architectural design of the synthesizer.

### 4.1 Autotuning

We use the synthesizer to autotune for a fast schedule. For example, sketch "*?hole*$_1$ ; parPre *?hole*" is underconstrained so the autotuner has freedom in choosing how to fill *?hole*$_1$. The synthesizer provides two correct choices:

*?hole*$_1 \in$ { **parPost**{w,h}, **parPost**{w} || **parPost**{h} }

Which schedule is faster depends on both the hardware and the size of expected trees. The first completion exposes more parallelism. However, on webpage-sized trees for multicore hardware, the overheads of a single traversal are high so the second completion is better. In general, the choice is not obvious because the selection of attributes for early traversals impacts the traversals possible later, and performance varies depending on hardware and trees.

Our autotuner design is simple. First, the developer provides input trees and hardware to test on. Second, the synthesizer enumerates all correct schedules. Finally, the autotuner compiles the schedules, profiles them on the inputs, and returns the fastest one.

### 4.2 Embedding the scheduling DSL in Prolog for first-class sketches and symbolic constraints

We implemented the synthesizer as a standard Prolog [6] relation. Doing so enables schedules to be unified with arbitrary Prolog programs rather than just sketches with holes. For an intuition, holes now simply map to the Prolog convention of using "_" for anonymous variables to automatically unify. Arbitrary Prolog programs can be used to further constrain them.

Our embedding treats schedule terms as first-class citizens that can be constrained with standard Prolog programs. For example:

```
Sched = [(T$_1$,A$_1$), recursive, (T$_2$,A$_2$)],
subset([x,y],A$_1$),
(T$_2$ = parPre ; T$_2$ = parPost)
```

The first line defines the schedule as a sequence of two traversals. The second line requires that the first traversal solves for, at least, x and y attributes. It demonstrates that schedules are first-class values. The final line specifies that the second traversal is either a parallel preorder traversal or a parallel postorder traversal. These last two lines demonstrate symbolic constraints beyond simple holes.

Our synthesizer provides constants **recursive** (";"), **parPre**, and **parPost**. Prolog provides operators [ ], ( ), subset, =, ; (disjunction), and "," (conjunction). Furthermore, it intreprets identifiers with capital first letters as variables to unify (e.g., Sched, $\mathrm{T}_1$, $\mathrm{A}_1, \ldots$).

We implemented the DSL with two techniques. First, we implemented the synthesizer as a search in Prolog. This enables us to reuse Prolog's unification algorithm and, for the sketching language, arbitrary Prolog programs. Second, as in Section 3, the algorithm unifies each prefix rather than just the final schedule.

### 4.3 Parallelism Debugging

We reuse the sketching language as an interface for three schedule debugging tasks: exploring, analyzing, and testing schedules.

***Exploring underconstrained schedules*** The synthesizer can show different completions to the programmer. This provides concrete understanding of the space of opportunities.

***Analyzing bottlenecks*** Our embedding in Prolog enables queries that analyze one or more schedules. The insight is that each schedule is a first-class Prolog value. For example, the diff of two schedule expression trees provides a lightweight change impact analysis. If a set of attributes can be synthesized as a **recursive** traversal but only a subset as **parPre**, the programmer knows that there is a problematic dependency for attributes in the difference of the two sets.

***Testing schedules*** The synthesizer can be used to try new schedule ideas. If the sketch cannot be filled in, the synthesizer returns an informative error. In particular, it describes the "first" unsynthesizable tree traversal in the sketch, meaning the bottom leftmost failing traversal in a failing schedule's expression tree.

We found support for exploring, analyzing, and testing schedules enabled a productive workflow. Early on, a developer examines possible schedules and fixes a subset using a sketch. Then, as ideas form on how to remove bottlenecks in the schedule, a developer can iterate between checking and analyzing schedules. Meanwhile, development focused on the functional specification can rely on the checker to detect any changes that violate the parallel schedule.

### 4.4 Loops, Interfaces, and Traits for Attribute Grammars

Our experiences with document layout languages led to extending attribute grammars with several features: recurrence relations (loops), information hiding (interfaces), and code reuse (traits).

The synthesizer does not need to be modified to support these constructs. Instead, we reduce the scheduling the extensions to that of plain attribute grammars. Subsequent code generation inverts the reduction to recover the used features. The following high-level implementation strategies illustrate how this can be done:

***Loops*** A tree may have a statically unbounded number of children. We support recurrence relations for computing over them. To demonstrate synthesis over non-trivial array expressions, consider an intentionally obfuscated way to count the number of chidlren:

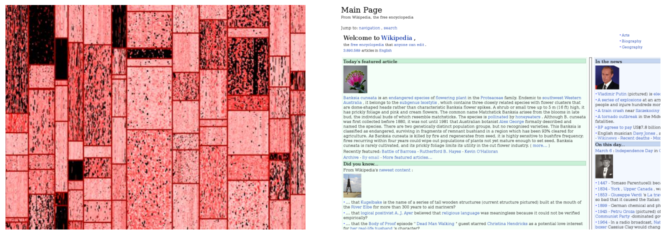
```
HBOX_0 → HBOX_1* {
  HBOX_0.numChildren = HBOX_1*[last].rollLen1;
  HBOX_1*[init].rollLen1 = 0;
  HBOX_1*[i].rollLen1 = HBOX_1*[i-1].rollLen2 + 1;
  HBOX_1*[init].rollLen2 = 0;
  HBOX_1*[i].rollLen2 = HBOX_1*[i-1].rollLen1 + 1 }
```

The synthesizer finds a loop interleaving rollLen1 and rollLen2 calls.

Loops resemble the uniform recurrence relations of Karp et al. [7, 14]. Ours are more expressive in that loops support escaping. For example, global sequential dependencies require **recursive** traversals to recur mid-iteration. We restrict the language of array indices to guarantee that the schedule for a few unrolled steps of the recurrences generalizes to recurrences of any length.

***Traits*** Traits support code sharing. They share declarations across productions. For example, trait paintRect can be added to any production with attributes $\{x_V, y_V, w_V, h_V, fill_V\}$ via "V → W (paintRect) {. . . }". Traits are implemented with macros.

***Interfaces*** Interfaces support information hiding. The programmer associates every non-terminal with a set of attributes: its public interface. For production $V \rightarrow W$, semantic functions may read and write any attribute of $V$ but only interface ones of $W$.



**(a) Votes** Five-pass parallel treemap visualizing Russian election data. **(b) CSS** 9-pass parallel CSS engine run on Wikipedia.

**Figure 10: Visualizations rendered with two grammars**

| 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 11: 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.

We found that information hiding provides an opportunity for optimization. The synthesizer only needs to schedule interface-level attributes. The availability of the rest can be inferred locally.

## 5. Evaluation

We evaluated the key aspects of our synthesizer. First, our algorithm synthesizes one or more schedules in a reasonable amount of time. Second, by exposing traversal structure information to a parallel runtime, we see 2-7X speedups over other approaches. Third, we describe the ability to add new traversal types. Finally, we performed two case studies of being able to apply our synthesizer: a multicore implementation of the CSS webpage layout language achieves 3X speedups on 4 cores, and a GPU implementation of a visualization of the 2011 Russian elections supports real-time interactions with over 96,000 polling stations.

### 5.1 Synthesis Speed

We measured the time to synthesize several attribute grammars:

1. **HBOX++** `H-AG` extended with more node types and styling
2. **Spiral** A radial visualization of space taken in a file system
3. **Votes** An interactive treemap of the 2011 Russian elections
4. **CSS** A CSS subset with floats, tables, and nested text

Figure 11 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 traversal sketch (`sketch`), takes less than 30 seconds. The exception was CSS, which we discuss in its own subsection 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 reached.

| | Total speedup | | | | Parallel speedup | | |
|---|---|---|---|---|---|---|---|
| | Cores | | | | 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 | | |

**Figure 12: 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.

### 5.2 Parallel Speedups From Structured Traversals

By statically exposing traversal structure (e.g., **parPre**) to our code generators, we observe sequential and parallel speedups. Our code generator performs pointer compression [17] and tiling [12] to improve sequential and parallel memory access, and beyond the scope of this paper, a new semi-static variant of work stealing to schedule tiles. We compare to using the Intel's TBB [23] dynamic task scheduler that performs work stealing [2] over the tiles.

For random 500-1000 node documents in the `hbox++` language, we saw 6.9X parallel speedups on 8 cores with our custom scheduler (`FTL`). For TBB, we saw slowdowns until 8 cores. The results are consistent across hardware (Figure 12). Finally, we also report sequential speedups of 1.2X-1.4X due to the memory optimizations, yielding a combined superlinear speedup of 9.3X on 8 cores. Static scheduling yielded significant speedups.

### 5.3 Autotuning

We evaluated schedule autotuning speedups for `hbox++` (`laptop`):

***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 ($\sigma/\mu$) 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.

***Greedy vs. non-greedy*** Our schedule enumeration is not exhaustive because of the greedy heuristic, and therefore may miss fast schedules (Section 3.2). 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++` .

### 5.4 Adding New Scheduling Primitives

Adding new scheduling primitives is simple. The average length of our primitives is 61 lines of commented Prolog code. For example, our nested traversal primitive was actually conceived of late into the CSS case study and took only 82 lines of code.

As another example, consider adding the scheduling primitives of spawn and join [2]. Their use is theoretically possible by generalizing the approach of FNC-2 [13] to extend our **recursive** traversal. Explicit schedules would include spawn and join points, which requires extending the syntactic enumerator. Any recursion point is a legal spawn but, if a statement depends on an attribute set by a preceding spawn, a join must be scheduled beforehand. To check this property, we can modify topological sorting ($\texttt{reachable}_{\texttt{taskRecursive}}$) to check reads. We did not add this feature because, as Jourdan [13] report, its reliance on dynamic scheduling suggests poor strong scaling (Section 5.2).

| | | 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 |

**Figure 13: Parallel CSS layout engine**. Run on a 2356 Opteron.

### 5.5 GPU Case Study: Interactive Treemap of Elections

We examined synthesizing GPU-accelerated code for an interactive and animated treemap of the 2011 Russian legislative elections for exploring anomalous voting activity. To run it, we created a GPU backend that generates level-synchronous breadth-first tree traversal [20] in OpenCL. Figure 10 (a) shows a real-time rendering of the entire data set of 94,601 polling stations. A surprising result was that parallelization was fully automated: the visualization programmer did not know the traversal schedule.

On a laptop-grade GPU (GeForce GT 650M with 384 cores), we measured end-to-end performance on three data sets: 10,000, 100,000 and 1,000,000 nodes. They achieved 27.6 fps, 27.6 fps, and 4.5 fps, respectively. We compared to running in our JavaScript backend, representing another high-level language. On Chrome 21.0 and `laptop`, JavaScript ran at most 500 nodes at 27 fps. Finally, we compared layout time between the GPU a multicore CPU (`server`). The laptop GPU had a 1.6X speedup over the server for layout, and not measured, directly invoked rendering without transferring layout data.

### 5.6 Multicore Case Study: CSS Webpage Layout

We synthesized a multicore implementation of the CSS language for webpage layout. The official CSS standard [3] is informal, written assuming an implementation using sequential tree traversals, and notoriously hard to implement even without considering parallelism. Further challenging supporting CSS is its many interacting features. For example, tables and nested text were not discussed in previous work [21] and required a new schedule.

Figure 10 (b) shows our grammar's result for laying out Wikipedia. We implemented features suggested by Mozilla developers (floats, automatic tables, nested blocks and inlines), as well as others seen in our tested websites, such as margins, padding, borders, relative positioning, and lists. Additional features, such as clearance and generated content, are ongoing targets for concurrent work in a tested, mechanized, and verified semantics of CSS.

For parallelization, the developer sketched to explore scheduling ideas. Usefully, edits to the grammar were checked for unsatisfiable dependencies and against the parallel schedule sketch. Our current schedule is a sequence of 9 parallel traversals, including one nested traversal with a sequential region for nested text. With a sketch similar to this description, synthesizing the schedule takes one minute. During development, the programmer typically experienced even faster synthesis times. He would manually incrementalize by only synthesizing edited classes.

We report a 3.1X speedup on layout (Figure 13). Our case study presents two milestones: the largest executable yet declarative specification of CSS and the first case of strong scaling.

## 6. Related work

***Parallel document layout*** Parallel layout is a difficult challenge. Browsers load independent resources in parallel, which can be used for parallel layout by decomposing a page into independent units. Concurrent work by Mai et al. [19] lays out a page on a proxy server, rewrites it as visually disjoint documents, and sends them to a client incorporating shared memory parallelism optimizations similar to those of Meyerovich and Bodík [21] for parallel rendering. We instead parallelize by optimizing just the layout engine.

Brown [4] propose applying task parallelism, which Meyerovich and Bodík [21] implement using Cilk [2] and TBB [23] to weakly scale a CSS subset. Burckhardt et al. [5] likewise apply task parallel Revisions. We achieve strong scaling and on a bigger subset.

***Attribute grammars*** Attribute grammars, first introduced by Knuth [16] to define language semantics, are tractable for static analysis and optimization. Saraiva and Swierstra [24] specify non-automatic HTML table layout with attribute grammars and Meyerovich and Bodík [21] first examine CSS.

Kastens [15] presents a sequential schedule synthesizer based on a global dependency analysis. Jourdan [13] surveys parallel attribute grammar evaluators that largely extend this idea. Most similar to our work are systems that find completely independent sets of attributes, corresponding to parallel composition ("||"), and those that treat each node as a dynamically schedulable unit. The former is too coarse-grained to expose significant parallelism, while, as reported by Jourdan for FNC-2, the latter rely upon runtime schedulers such as work stealers and thus only weakly scale.

***Synthesis*** Data structure synthesis is an old problem. Early on, Low [18] examined multiple set implementations for an ALGOL-60 variant. Recent work by Hawkins et al. examines tuning over data structures for relational code [11]. Our paper leaves decisions of how to implement tree data structures to our optimizing backends and instead focuses on how to determine what tree traversal types characterize the computations.

Our autotuning compiler backend is similar to the ATLAS [26] framework for linear algebra in that we tune parameters such as block size to optimize for a particular device. Autotuning is actively being applied to further domains, such as work in stencils [8] by Datta et al.: our backend examines the case of traversals over trees.

Similar to our schedule autotuner, FFTW [10] and PetaBricks [1] select algorithms, not just parameters. However, programmers must state what algorithms to try in those systems. Our synthesizer infers what is possible. Elixir [22] infers dynamic task scheduling optimizations for a single for-loop. Assuming extensions for structured traversals, Elixir might replace our code generators for concrete schedules. Hypothetically, our system would synthesize schedule "**parPost**{w,y};**parPre**{x,y}" and could then ask Elixir for dynamic task scheduling optimizations for traversal "**parPre**{x,y}".

Finally, our language extension for programming with holes, which we use for scheduling, is inspired by Sketch [25].

## 7. Conclusion

We presented a synthesizer that can schedule an attribute grammar as a composition of parallel tree traversals. The synthesizer's enumerate-and-check design simplifies adding new traversal types and extensions such as debugging. For $O(n^3)$ synthesis and practical autotuning, we optimized its algorithm. Furthermore, to enable control of schedules, we introduced a declarative language and extended it with sketches. Finally, we successfully performed two non-trivial case studies: multicore parallelization of a large fragment of the CSS webpage layout language and GPU acceleration of a Russian election animation. Put together, we demonstrate a path to more productive and effective parallel programming.

## Acknowledgments

We thank anonymous reviewers for presentation suggestions. Adam Jiang and Edward Lu contributed to early versions of this work. Discussions with Krste Asanovic, Boris Zbarsky, Robert O'Callahan, Wolfram Schulte, Todd Mytkowicz, David Sheffield, Shoaib Kamil, Scott Beamer, and others helped guide our case studies.

## References

[1] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *PLDI'09*, June 2009.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP'95*, pages 207–216, 1995.

[3] B. Bos, H. W. Lie, C. Lilley, and I. Jacobs. Cascading style sheets, level 2 CSS2 specification, 1998.

[4] H. Brown. Parallel processing and document layout. *Electron. Publ. Origin. Dissem. Des.*, 1(2):97–104, 1988.

[5] S. Burckhardt, D. Leijen, C. Sadowski, J. Yi, and T. Ball. Two for the price of one: a model for parallel and incremental computation. In *OOPSLA'11*, pages 427–444, 2011.

[6] A. Colmerauer. An introduction to Prolog III. *CACM*, 33, July 1990.

[7] A. Darte and F. Vivien. Revisiting the decomposition of Karp, Miller and Winograd. In *ASAP*, pages 13–25, 1995.

[8] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC'08*.

[9] S. Dubey. AJAX Performance Measurement Methodology for Internet Explorer 8 Beta 2. *CODE Magazine*, 5(3):53–55, 2008.

[10] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[11] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Data representation synthesis. In *PLDI'11*, June 2011.

[12] F. Irigoin and R. Triolet. Supernode partitioning. In *POPL'88*.

[13] M. Jourdan. A survey of parallel attribute evaluation methods. In *Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 234–255. Springer Berlin / Heidelberg, 1991.

[14] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, July 1967.

[15] U. Kastens. Ordered attributed grammars. *Acta Informatica*, 1980.

[16] D. E. Knuth. Semantics of context-free languages. *TOCS*, 2(2):127–145, June 1968.

[17] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *PLDI'05*.

[18] J. Low. Automatic data structure selection: an example and overview. *CACM*, 21(5):376–385, 1978.

[19] H. Mai, S. Tang, S. T. King, C. Cascaval, and P. Montesinos. A case for parallelizing web pages. In *HotPar'12*, 2012.

[20] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In *PPOPP '12*, pages 117–128, 2012.

[21] L. A. Meyerovich and R. Bodík. Fast and parallel webpage layout. In *WWW'10*, pages 711–720, 2010.

[22] D. Prountzos, R. Manevich, and K. Pingali. Elixir: a system for synthesizing concurrent graph programs. In *OOPSLA '12*, October 2012.

[23] J. Reinders. *Intel threading building blocks*. O'Reilly, 2007.

[24] J. a. Saraiva and D. Swierstra. Generating spreadsheet-like tools from strong attribute grammars. In *GPCE'03*, pages 307–323, 2003.

[25] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS-XII*, pages 404–415, 2006.

[26] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.