FTL: Synthesizing a Parallel Layout Engine

Abstract

Visual layout languages are important for running web browsers on mobile devices, but they are difficult to design, implement, and optimize. Our solution is the Fast Tree Language (FTL) *layout engine generator*. Given an attribute grammar defining the language's semantics and layout instances to profile, FTL outputs the first strongly scaling parallel layout engine.

Parallelizing the small tree traversals that characterize layout solving is challenging. We show how to combine existing techniques, and where needed, present new ones. First, to assist designing parallelizable specifications, we introduce a declarative query and constraint language over grammar evaluator schedules. Next, to help find the fastest evaluator schedule, we present how to synthesize a variety of attribute grammar evaluation orders. Third, as memory bottlenecks prevent scaling, FTL autotunes over data layout optimizations. Finally, for balanced, locality-aware, and low-overhead scheduling, FTL partitions the document tree among cores using a novel semi-static approximation of work stealing.

We measure strong scaling within 4% of ideal on a single-socket quad-core device, and, including data layout optimization, total speedup of 5.2x. On 2 sockets, total speedup is 9.3x and within 14% of the ideal.

1. Introduction

Document layout implementation is a long-studied [7, 11, 31] problem that is important for consumer products. For example, studies at Facebook, Google, Microsoft, and Amazon [44] measure that decreasing web page load time by 2s improves user engagement enough to increase annual revenue by 1-2%. Professional designers must therefore balance performance with rich features and productivity. To ease this burden, we present techniques for designing and implementing parallel layout languages.

Characterizing the performance bottleneck, Anonymous [4] and Meyerovich et al. [27, 39] find that loading webpages is CPU-bound and layout logic accounts for 15-22% of the CPU time. The trade-off between performance, productivity, and feature set is pronounced for mobile devices. Loading the same popular webpages on a 1st generation iPhone as on a MacBook Pro laptop [27, 39] suffers from a magnitude drop in performance. A developer porting a laptop application to a mobile device will manually optimize inefficient features or even switch to more imperative and low-level languages.

Correctly predicting that commodity mobile architectures would be parallel, Jones et al. [27] propose architecting a browser as concurrent components and exploiting fine-grained parallelism within each component. Parallel browser components have since been studied [4, 20, 39] and deployed [6].

Layout engines, however, are still not parallel. Brown [11] proposes a task parallel decomposition; implementing it, Meyerovich and Bodík [39] and Burckhardt et al. [12] only report weak scaling. Parallel layout of modern documents remains an open problem because small pointer-chasing tree traversals are challenging.

We address the layout language implementation problem by introducing a level of abstraction: the Fast Tree Language (FTL) *layout engine generator*. Given a layout language declaratively specified as an attribute grammar [5, 30, 39, 43], FTL generates a parallel implementation with a tuned data layout and scheduler.

FTL extends existing static [29] and parallel [28] attribute grammar evaluation techniques. We briefly introduce FTL's design and the technical contributions in each component:

Grammar specification and debugging language (Section 3): Designers either specify a layout language as an attribute grammar or use one of our higher-level languages [2, 5] that compile into one. FTL then statically finds an implementing sequence of tree traversals called a *visit order* [28, 29]. Each traversal follows an optimizable control pattern, such as *parallel top down*.

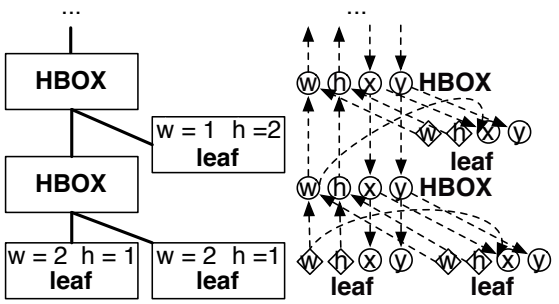
As traditional in attribute grammar compilation [29], FTL assists grammar designers by statically checking that every well-formed input tree has a well-defined layout and that the asymptotic complexity of evaluation is bounded. At this point, FTL can also generate an inefficient layout engine that a designer can then test against input documents.

Designing grammars for parallelization is still difficult so FTL provides further support. First, FTL supports constraints on visit order, such as to disallow non-reentrant foreign function calls within a parallel traversal. Debugging support is also important: FTL supports queries over visit orders. For example, a designer can request the set of parallel visit orders, or query for the dependency that breaks a parallel order she expected to work. Constraints and queries are phrased as standard Prolog [14] predicates.

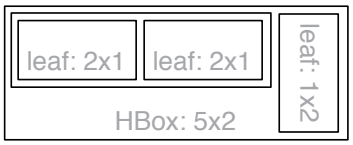
Querying and constraining visit orders is a new approach to designing grammars, as is matching them to optimizable patterns.

Evaluator synthesizer and tuner (Section 4): FTL's synthesizer generates a variety of visit orders for an input grammar. We use the synthesizer for our first autotuning optimization: FTL profiles different visit orders and picks the fastest.

The space of visit orders is large, so we present an optimized search for them. First, as many visit orders are similar, it skips redundant pattern matching calls by memoizing them and incrementally approximating their weakest precondition [17]. Second, it quickly finds a valid visit order matching a concrete sequence of patterns (e.g., a top-down traversal followed by bottom-up) by monotonically weakening an assume/guarantee proof [40]. Finally,



(a) Input tree (b) Dynamic data dependencies



(c) Output visualization with (w,h) labels

Figure 1: Tree input, data dependencies, and visualized output.

it prunes the search using a greedy heuristic [29] that evaluates attributes in the first traversal possible.

We found synthesizing and autotuning visit orders to be effective new approaches. For a CSS subset run on sequential hardware, performance between the fastest and slowest differs by 32%.

Code generator with data layout tuning (Section 5): Given a synthesized sequence of tree traversals, a code generator outputs an executable implementation integrated with a parser and a renderer.

Code generation also emits data layout optimizations because traversing small trees makes memory access costly. E.g., pointer compression replaces 64bit pointers with 8bit relative offsets and subtrees are allocated in blocks for spatial locality. Our second autotuning optimization is over the set of these code generated optimizations, including their parameters. For example, block size depends on the cache size.

We show our data layout optimizations, such as compression, are important for computations on small trees. Without them, speedup on 4 cores is 30% from perfect scaling. With them, it reaches 4% of ideal. We also observe a 26% sequential speedup.

Semi-static task scheduling with a work stealing approximation (Section 6): At runtime, FTL's semi-static task scheduler examines the document tree to partition work among cores. To amortize task scheduling costs across many nodes and improve locality, the partitioner splits the tree into contiguous blocks. To load balance, the partitioner precomputes the assignment of task blocks to cores using a heuristic that *simulates* work stealing. As with visit order selection and data layout optimization, we autotune over scheduler parameters.

Existing semi-static heuristics we tested were not load balanced, while using dynamic work stealing to load balance exhibited no speed up at all. Our use of work stealing as a heuristic helps connect these approaches.

2. Background

In this section, we review layout solving. A layout engine solves constraints such as for the size, position, and color of a visual element. To efficiently do so, it solves all of them over a statically determined visit order. The optimization problem is to pick the most efficient order and further optimize individual traversals.

```
1  class Pass: extends RecursiveSequential {
2    visit(HBOX n) {
3      n.BOXI[0].y = n.y;
4      n.BOXI[1].y = n.y;
5      n.BOXI[0].x = n.x;
6      n.BOXI[0].visit(this);      //recur
7      n.BOXI[1].x = n.x + n.BOXI[0].w;
8      n.BOXI[1].visit(this);      //recur
9      n.h = max(n.BOXI[0].h, n.BOXI[1].h);
10     n.w = n.BOXI[0].w + n.BOXI[1].w;
11   }
12   /* ... visit S, BOXI, leaf ... */
13 }
14 new Pass().traverse(tree); //sequential internally
```

Figure 2: Sequential visitor

```
1  class Pass0 : extends TopDownParallel {
2    visit(S n) { n.BOXI.x = 0; n.BOXI.y = 0; }
3    visit(HBOX n) { n.BOXI[0].y = n.y; n.BOXI[1].y = n.y; }
4    visit(LEAF n) { n.w = 10; n.h = 10; }
5    visit(BOXI n) { ... /* compute y */ }
6  }
7  class Pass1 : extends BottomUpParallel {
8    visit(HBOX n) {
9      n.w = n.BOXI[0].w + n.BOXI[1].w;
10     n.h = max(n.BOXI[0].h, n.BOXI[1].h);
11   }
12   visit(BOXI n) { ... /* compute w, h */ }
13 }
14 class Pass2 : extends TopDownParallel {
15   visit(HBOX n) {
16     n.BOXI[0].x = n.x;
17     n.BOXI[1].x = n.x + n.BOXI[0].w;
18   }
19   visit(BOXI n) { ... /* compute x */}
20 }
21 new Pass0().traverse(tree); //parallel internally
22 new Pass1().traverse(tree); //parallel internally
23 new Pass2().traverse(tree); //parallel internally
```

Figure 3: Parallel visitors

As input, a layout engine takes a tree where attribute values of some nodes are known. For example, in Figure 1(a), a layout designer provides a tree with predefined widths and heights on leaf nodes. Each node corresponds to a visual element: the engine must compute any remaining attributes, such as absolute x and y coordinates for all nodes (Figure 1(c)). The layout logic depends on the layout language. In this case, the width w of an intermediate horizontal box node (HBox) is the sum of its children widths (BOXI[0].w and BOXI[1].w) and its height h is the maximum of their heights. Dynamic data dependencies restrict the order in which attributes may be evaluated (dashed lines in Figure 1(b)).

An efficient layout engine computes all attribute values over a statically determined visit order. For a simple language of horizontal (HBox) and leaf (Leaf) boxes, we must already choose between sequential and parallel traversal patterns:

- **Sequential:** One recursive pattern instance can sequentially solve any tree in the language. Figure 2 shows how using a *visitor* [22]. Upon reaching an HBox node, the visitor sets y attributes of the children and x of the left child, recurs on the left child, uses the result to set the x of the right child, recurs on the right child, and then uses both results to set w and h.
- **Parallel:** Figure 3 depicts an alternative visit order: a sequence of 3 topologically parallel traversals. A top-down traversal com-

```
HBox : BoxI
 w = 0 .. _ + children[i].w
 h = 0 .. max(_, children[i].w)
 children[i].x = x .. _ + children[i - 1].w
 children[i].y = y
```

Figure 4: Encoding of the HBox widget in language β.

```
S → BOXI { BOXI.x = 0; BOXI.y = 0 }
BOXI →
  HBOX
  {w = HBOX.w; h = HBOX.h; HBOX.x = x; HBOX.y = y}
| leaf
  {w = leaf.w; h = leaf.h; leaf.x = x; leaf.y = y}
HBOX → BOXI BOXI {
 BOXI[1].x = x
 BOXI[2].x = x + BOXI[1].w
 BOXI[1].y = y
 BOXI[2].y = y
 h = max(BOXI[1].h, BOXI[2].h)
 w = BOXI[2].w + BOXI[2].w
}
leaf {w = 10; h = 10}
```

Figure 5: Attribute grammar for a language of horizontal boxes.

putes *y* attributes, then a bottom-up traversal computes *w* and *h* attributes, and finally another top-down traversal computes *x* attributes. To evaluate in parallel, each node visit is a task dependent on its parent in a top down traversal and dependent on its children in a bottom up one.

Many other visit orders also solve all attributes. Part of the optimization problem is to pick the most efficient one.

Optimizing a pattern instance is difficult, in part, because the computation is small. Meyerovich and Bodík [39] measure that, on average, solving layout takes 84ms when loading pages on a 2.4GHz laptop. Treating every visit to a node as a dynamically scheduled task, for example, would have high overhead. Our challenge is in finding the fastest visit order and efficiently implementing each pattern.

3. Designing Parallel Grammars using Synthesis

FTL assists designing parallelizable attribute grammars. It statically checks an attribute grammar is well-defined [28, 29] and automatically generates a parallel implementation. Designing languages with a particular intended parallelization scheme is difficult due to non-local reasoning, so we also introduce a declarative language to query and constrain visit orders. We explain these features by demonstrating how a designer can extend the HBox language while still supporting parallel evaluation.

3.1 Specifying an attribute grammar

FTL computes over a low-level attribute grammar representation. Out of scope of this work, we compile 2 constraint languages to attribute grammars: α [5], a bidirectional language, and β [2], an object-oriented one. Figure 4 specifies the HBox widget in β.

Figure 5 fully specifies a language of horizontal boxes as an attribute grammar. Terms outside of the braces define the tree structure as a context free grammar. A BOXI node has either a leaf terminal or HBOX non-terminal, where an HBOX node has 2 BOXI children. This example grammar implicitly encodes β's class system: BOXI encodes an interface implemented by leaf and HBOX.

Terms inside the braces of each production specify the absolute coordinates (x, y) and sizes of nodes (w, h). For example, the start node S positions the topmost box in the tree at the top left coordinate $(0, 0)$. Likewise, the width w of an instance of HBOX is the sum of the widths of its two child nodes. Note that a programmer does not specify when to compute the various attribute values.

3.2 Checking, testing, and inspecting an evaluator

FTL implements several traditional attribute grammar compiler techniques [29] to support the edit/compile development cycle.

First, FTL checks if a language is well-defined and permits evaluation that is $O(n)$ in the number of nodes. Satisfying these conditions when defining features that can be composed like HBOX and leaf nodes is difficult. For example, the CSS specification is ambiguous: some of the ambiguities have been found and surface as cross-browser incompatibilities, and it is not clear if all the ambiguities have been found. [10, 39] Natural formulations of seemingly simple features like percentage and automatically shrinking boxes are fixed point computations, but this is often incorrect due to undesirable visualizations of the singularities [39] and poor performance [45]. FTL checks that a syntactically well-formed input tree has exactly 1 solution and that it can be found in $O(n)$.

If an evaluator is possible, FTL generates an implementation in C++ or HTML5, which can be tested. The sample language (Figure 5) passes the checks, so FTL generates both C++ and HTML5 implementations. The programmer can test either on an input document that is syntactically HTML and CSS.

The designer may be interested in checking the evaluator's visit order has the correct performance properties, e.g., performs only a few parallel traversals, and thus may inspect a summary:

```
Schedule=[
 (td, [(s,boxi,x),(s,boxi,y),
       (hbox,boxi1,y),(hbox,boxi2,y),
       (boxi,hbox,y),(boxi,leaf,y)]),
 (bu, [(hbox,self,w), (hbox,self,h), ...]),
 (td, [(hbox,boxi1,x),(hbox,boxi2,x), ...])]
```

The output corresponds to the visit order for Figure 3. It is a sequence of (p_i,t_i) pairs, where p_i is the pattern followed on traversal i and t_i is a list of attributes computed during it. Above, FTL finds the parallel pattern sequence P = [td, bu, td], where td is a top down traversal, bu is bottom up, and both are parallel internally.

3.3 Querying parallel behavior

FTL provides a Prolog [14] interface for declaratively debugging a grammar for parallel visit orders. Consider extending the parallel HBox language to support vertical boxes:

```
VBOX → BOXI BOXI {
 BOXI[1].y = y
 BOXI[2].y = y + BOXI[1].h
 BOXI[1].x = x
 BOXI[2].x = x
 w = max(BOXI[1].w, BOXI[2].w)
 h = BOXI[2].h + BOXI[2].h
}
```

The designer reasons locally in changing the specification, but the impact is global. In particular, adding the VBOX widget breaks the original parallel schedule P = [td, bu, td]. The first pass can no longer solve y attributes: VBOX nodes require unavailable h attributes. The local specification change requires a global refactoring of the evaluator that moves terms dependent on y attributes.

FTL can summarize all valid visit orders to reveal the impact:

```
?- synthesize(Schedule, (_, [])),
|  findall(P_i, member((P_i,_), Schedule), P).
```

The first line uses FTL's synthesize predicate to unify variable Schedule with any valid order, similar to the original output summary, and the second line simplifies it as a sequence P of p_i values using standard Prolog. All possible unifications are output.

The search quickly finds P = [bu, recursive], P = [recursive], and P = [bu, td]. The original 3-pass parallel visit order is not possible, though the fully parallel order [bu, td] is.

The designer can compare the original 3-pass visit order to the new [bu, td] one to understand which attribute computations moved. She might ask, for sequence P = [bu, td], what are the corresponding t_1 and t_2:

```
?- synthesize([(bu,T_1), (td,T_2)], (_, [])).
```

The answer changes from the original unextended grammar: y attribute computations for HBOX and BOXI move to the last traversal.

To avoid refactoring, the implementer may examine the viability of keeping y attribute computations in the first traversal:

```
?- synthesize([(td,T_1) | _], (Progress, Unsolved)),
| member( (hbox,boxi1,y), T_1),
| member( (hbox,boxi2,y), T_1).
```

Variable Progress is unified with the list of (p_i, t_i) pairs of all leading correct traversals, and Unsolved with any remaining attributes, namely those from subsequent traversal. In this case, as hbox.box1.y cannot be solved in the requested first traversal, there is no initial progress so the debug output shows all attributes remain unsolved. A specification-level correction is therefore required to enable initial computation of y attributes, and further FTL functions can help reveal the problematic dependency on attribute h.

Overall, we see FTL assists finding and debugging the parallelism in an attribute grammar.

3.4 Constraining parallelism and order in evaluators

The synthesize predicate also helps satisfy embedding concerns.

Consider the following two constraints: y attributes must be computed in the second traversal, which must be sequential. The first constraint on order is common to layout frameworks that dynamically compose with third party components. The second constraint enables safely calling non-reentrant libraries. Both constraints are beyond canonical attribute grammar specifications.

The synthesize predicate can be used to express both constraints:

```
?- findall((Prod,Vert,y), attrib(Prod,Vert,y), As),
| subset(As, T_2),
| synthesize([_, (recursive, T_2) | _], (_, [])).
```

The only novel FTL construct in the query is predicate attrib, which is a relation naming production attributes. The query succeeds, returning a visit order with the desired staging of sequential and parallel code. Consider manually adding these constraints to a hard-coded implementation: significant global reasoning and refactoring would be involved instead of these 3 declarative lines!

4. Grammar Compilation as Synthesis

Our synthesizer must find every visit order (p, t) for a grammar, but there are many. This section presents an optimized search that reuses partial solutions, monotonically finds a valid t for a fixed p, and employs a greedy heuristic. Once generated, the grammar design language of the previous section can support queries over these pairs and our code generator in the next section will convert a pair into an executable evaluator and autotune over them.

4.1 Evaluator synthesis problem definition

The input is an attribute grammar $ag = (G, A, F)$ and set of attribute compiler indicator functions $\chi = \{\chi_{\text{td}}, \chi_{\text{bu}}, \chi_{\text{recursive}}, \ldots\}$ that recognize different patterns. The goal is to find all combinations of using different patterns in χ for different traversals.

The first part of the input definition is a standard attribute grammar. Tuple $G = (N, T, S, P)$ is a context free grammar with N the set of non-terminal nodes, T the terminal nodes, P the productions

```
1  INPUT: χ : { 𝔸𝔾×{Attrib}×{Attrib}→{Attrib} }
       × ag : 𝔸𝔾
2  OUTPUT: prefixes : { (χ_p, {Attrib}) }
3  partialPrefixes := { [ ] }
4  completePrefixes := ∅
5  ŵp := ∅
6  while pre := partialPrefixes.remove():
7    previous := { attrib | (p_i, t_i) ∈ pre and attrib ∈ t_i }
8    candidates := ag.A − previous
9    for χ_p ∈ χ:
10     t_i := candidates
11     ok := false
12     while ¬ok and t_i ≠ []:
13       if ŵp[t_i, p] ⊆ previous:
14         ok := true
15       else:
16         (ok, ϵ) := χ_p(ag, previous, t_i)
17         t_i := t_i − ϵ
18     if ok:
19       ŵp[t_i, χ_p] := ŵp[t_i, p] ∩ previous
20       if t_i = candidates:
21         completePrefixes.add(p @ [(p, t_i)])
22       else:
23         partialPrefixes.add(p @ [(p, t_i)])
24 return completePrefixes
```

Figure 6: Algorithm to find attribute grammar evaluators. Dynamically programs over prefixes, incrementally refines weakest precondition abstraction for χ, and performs greedy MSP heuristic.

over the nodes, and $S \in N \cup T$ some starting node. A and F label each node X with a set of attributes $A_X \in A$. Every production $(p = X_0 \to X_1 ... X_p) \in P$ is likewise extended so that attribute $X_i.a_j$ may be constrained as a function over some subset of the production's other attributes: these constraints are stored in set F.

Pattern compilation function $\chi_p : \mathbb{AG} \times \mathcal{P}(A) \times \mathcal{P}(A) \to \mathbb{B} \times \mathcal{P}(A)$ denotes an attribute grammar compiler that returns error messages on failure. It is similar to an actual attribute grammar compiler, except it does not generate an actual implementation. It returns true for an attribute grammar if all instances of the attributes in the last argument can be solved assuming preceding traversals solved all instances of the attributes in the second argument. If not all attributes in the last argument can be solved, χ_p returns an error message: a subset of the attributes that could not be solved. For example, $\chi_{\text{td}}(ag, \emptyset, A) = (false, \{VBox.y, \ldots\})$, meaning not all instances of VBox.y can be solved in an initial top-down traversal.

We define $solve(t, p)$ if and only if:

$$\bigcup t_i = A \qquad t_i \bigcap t_j \neq \emptyset \Leftrightarrow i = j \qquad \bigwedge \chi_{p_i}(ag, \bigcup_{j<i} t_j, t_i)$$

The first two properties define t as a disjoint partitioning of A. The third property asserts each step of the traversal sequences solves the intended attributes.

The problem is therefore to find all t, p such that $solve(t, p)$. A naïve brute force algorithm that exhaustively tries all different (t, p) sequences is too inefficient so the search must be optimized.

4.2 Reusing valid prefixes

We reuse knowledge of validity of prefixes of visit orders to examine longer ones that are suffixes.

The search algorithm generates increasingly long prefixes of solutions. This corresponds to lines 6, 9, 16, and 20-23 of Figure 6. Consider how to reach solution ($t_1 t_2$, bu td). We first test (t_1, bu) by calling $\chi_{\text{bu}}(ag, \emptyset, t_1) = (true, \emptyset)$. Upon success, we try completions ($t_1 t_2$, bu bu), which fails with $\chi_{\text{bu}}(ag, t_1, t_2) = (false, \epsilon)$, and ($t_1 t_2$, bu td), which succeeds as $\chi_{\text{td}}(ag, t_1, t_2) = (true, \emptyset)$

and satisfies $solve(t_1t_2, \text{td bu})$. Contrast with initial attempt $\chi_{\text{bu}}(ag, \emptyset, t_1)$ that $(false, \epsilon)$: prefix (t_1, bu) failed so there is no reason to try suffixes $(t_1t_2, \text{bu td})$ nor $(t_1t_2, \text{bu bu})$.

Calls to χ_p are expensive, so we avoid many invocations by checking against weakest preconditions [17]. For intuition, consider the original grammar for which we provide solution $solve(t_1t_2t_3, \text{td bu td})$. To test alternative $solve(t_{12}\, t_3, \text{recursive td})$ where $t_{12} = t_1 \cup t_2$, we need only invoke the compiler to check $\chi_{\text{recursive}}(ag, \emptyset, t_{12}) = (true, \emptyset)$. Check $\chi_{\text{td}}(ag, t_{12}, t_3) = (true, \emptyset)$ is avoided by fact $\chi_{\text{td}}(ag, t_1 \cup t_2, t_3) = (true, \emptyset)$ computed in the first solution. The insight is that changing the argument $t_1 \cup t_2$ with an equivalent or bigger set of previously computed attributes will not invalidate the pattern's applicability for computing t_3. Therefore, invocation $\chi_{\text{td}}(ag, t_{12}, t_3) = (true, \emptyset)$ is avoided by replacing it with superset check $t_{12} \supseteq t_1 \cup t_2$.

To employ the above reasoning, our algorithm uses and incrementally refines an approximation of the weakest precondition of satisfying each χ_p. The approximation $\widehat{wp}(t_i, p_i)$ for call $\chi_{p_i}(ag, previous, t_i)$ is one such that $previous \supseteq \widehat{wp}(t_i, p_i) \Rightarrow \chi_{p_i}(ag, previous, t_i)$. An invocation $\chi_{p_i}(ag, previous, t_i)$ is skipped if the weakest precondition guarantees it succeeds, namely, the attributes computed in previous traversals ($previous$) are a superset of $\widehat{wp}$ (line 14). We do not precompute the weakest precondition. Instead, every time a smaller subset of attributes is found sufficient to compute t_i for some pattern p_i than the current $\widehat{wp}(p_i, t_i)$ entry, the weakest precondition is refined (line 19).

4.3 Fast schedule testing

To prune how many suffixes $(t\ t_i, p\ p_i)$ are attempted for prefix (t, p), we perform a monotonic search over valid t_i. Not every t_i must be attempted. On failure output $\chi_{p_i}(ag, t, t_i) = (false, \epsilon_0)$, the compiler guarantees any set $t_i - \epsilon_0 \subset t_i' \subset t_i$ will also fail. The next tests are $\chi_{p_i}(ag, t, t_i - \epsilon_0)$, $\chi_{p_i}(ag, t, t_i - \epsilon_0 - \epsilon_1)$, etc. (lines 16-17), until the empty set is reached (line 12). The initial set of t_i candidates are any attributes not previously computed ("$candidates = ag.A - previous$", line 8 of Figure 6). Monotonically searching from this initial set finds all solving t_i subsets.

To implement a χ_p function for a particular pattern, we construct an iterative assume/guarantee [40] proof. Consider an initial bottom up traversal when it reaches an HBOX node on some arbitrary tree. The initial prospective t_i is all attributes, A. For the HBOX production, we therefore *assume* the traversal solved any production attributes that are specified by semantic functions outside of the HBOX production, and using these assumptions, *guarantee* semantic functions of the remaining HBOX attributes can be evaluated:

$$assm_{HBOX} = \{\text{BOXI[1].w, BOXI[2].w, BOXI[1].h, BOXI[2].h}\}$$
$$grnt_{HBOX} = \{\text{w,h,BOXI[1].x,BOXI[2].x,BOXI[1].y,BOXI[2].y}\}$$

The dependencies for computing w and h are satisfiable by assumptions in $assm_{HBOX}$, discharging part of the obligation. The remaining attributes depend on x and y, which are not computable from the assumptions. Evaluating χ_{bu} therefore fails with at least $\{BOXI.x, BOXI.y\}$.

Stated more generally, for each pattern, a proof for a grammar is modularly constructed by decomposing on productions. At the beginning of the traversal, all instances of attributes in t are solved. When a traversal reaches an instance of a production, t_i attributes of all other production instances previously visited are also assumed to be solved. The proof must then guarantee t_i attributes in the current production can be solved. Any attribute that cannot be guaranteed is put in ϵ: the assumptions are too weak to prove that it can be solved.

The search is pruned by removing uncovered error attributes from the prospective set, so we take care to return a large set. Note that similar reasoning for HBOX applied to the BOXI production rejects all other x and y attributes. An error is the union of failures from all productions rather than short circuiting on the first failure found. In the example, the search succeeds on just the second attempt, calling χ_p with w and h attributes.

4.4 Pruning similar results with the MSP heuristic

We use a greedy heuristic to quickly compute the fast and representative *minimal solving prefix* set MSP. The MSP heuristic prunes away searches for supersequences of any sequence $(t, p) \in MSP$ that was already found and ignores permutations of t.

Consider pattern sequences `[bu td]` and `[bu bu td]`, where the first pattern computes both w and h attributes in the first traversal, while the second pattern computes h in a new intermediate traversal. Delaying evaluation of h attributes in the first traversal allows introduction of an intermediate bu traversal: while there are theoretical performance benefits, we observed none in our implementation. We avoid these sequences by greedily computing as many attributes in a prefix as possible: the search for a t_i for some p_i stops at the first, and therefore biggest, valid t_i (line 12 of Figure 6).

Many t are possible for any sequence p, which we also ignore. First, implicit to our formalism, different permutations of t_i are ignored. For example, we observe minor benefit of swapping the order of statements in a visit function, such as swapping lines 1 and 2 of Figure 2. Second, we avoid repartitioning t for the same p. We did observe minor benefit from changing which t_i an attribute is computed (Section 7.3), though more so for final code generation than interactive grammar design or search for the fastest p. The MSP heuristic therefore ignores both variations in t by representing t_i as a set rather than an ordered list (lines 8, 10, 17) and greedily finding the first t for a given p.

5. Optimizing Data Layout

FTL performs data layout optimizations for 2 related purposes. First, to improve even sequential code performance due to the similarity of grammar evaluation to pointer chasing. Second, to improve scaling of parallel evaluation.

We found it useful to optimize locality, prefetching, and size. Data layout optimization algorithms and automation techniques for these tasks are well-studied in general but typically not in a unified and automated way. We implement multiple algorithms for each task and FTL autotunes over which to use and how.

In concurrent work, [3] we report a novel data layout optimization yielding a 3.8x speedup over techniques presented here. These are complimentary as they target performance not directly related to MIMD parallelism: branch prediction, SIMD evaluation, etc.

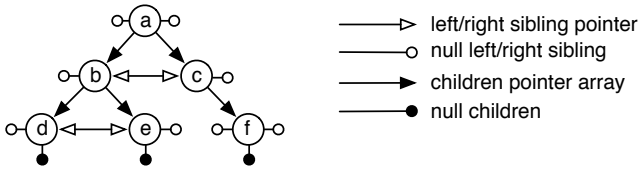
5.1 Compressed, high locality layout

FTL chooses from a variety of tree data layout options to improve spatial and temporal locality:

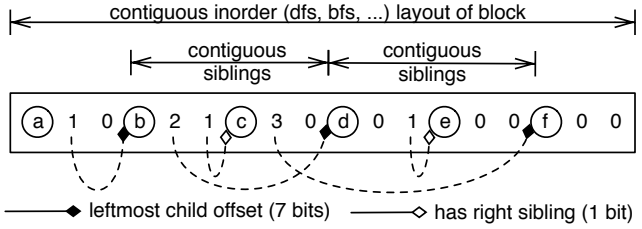
- C++ collections (the embedding language) or contiguous arrays
- depth-first or breadth-first orderings
- blocking [25] of subtrees
- aligned data or unaligned but more packed data

We also optimize tree traversal and structure access. The traversal order matches the layout order [13]. As a simple example of optimizing access for a particular data structure, inorder traversal of a block is performed by just successively incrementing a pointer.

FTL compresses the tree to better utilize bandwidth and decrease the working set size. Similar to Lattner and Adve [34], node references are encoded as relative offsets rather than native pointers. We also exploit the tree data structure to avoid some pointers.



(a) Naïve pointer-based tree representation



(b) Compressed tree encoding

Figure 7: Pointer-based (a) and relatively indexed (b) tree representation. Compressed uses 96% fewer bits to encode the structure of the depicted tree on a 64-bit architecture.

For example, as siblings are adjacent in certain encodings, we do not need sibling pointers. Furthermore, as there are typically few siblings, instead of a counter of number of children (or siblings), we use an isLastSibling bit. Figure 7 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.

5.2 Prefetching

FTL supports several forms of prefetching to avoid waiting on data fetches. First, as the above optimizations match data access patterns with the data layout, hardware prefetchers can automatically predict and prefetch data. Second, FTL also inserts explicit prefetch instructions as part of the traversal. Finally, runahead processing [18] 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.

5.3 Code generation and autotuning

Which optimizations to use and how is not obvious, so FTL incorporates them by autotuning. We implement the optimizations generically so grammar designers do not need to manually use them.

Autotuning supports optimizations sensitive to the input grammar, hardware configuration, and other optimizations. For example, when blocking, block size is sensitive to both the grammar and hardware cache sizes. Likewise, while more threads are generally better, we observed slowdowns on different schedules on all devices examined when trying different thread count increases.

The space of autotuning configurations is large and evaluating a individual configuration is slow enough that brute force exploration is too expensive. We currently manually preselect a set of optimization combinations and bound parameter ranges. E.g., we always use a blocked layout on parallel architectures as our parallel optimizations require it, and do not try many more virtual threads than hardware threads available. Finally, we stage autotuning for fast visit orders (Section 4) to occur before autotuning the data layout of selected variants.

Adding an optimization to FTL is generally a combination of modifying the visit-order-to-C++ backend and adding new macros or libraries to the C++ runtime. For example, pointer compression in the style of Lattner and Adve [34] is a 16 line change to the code generator and 200 to the runtime. Each is written to be conditionally compiled according to a compilation constants file or controlled by a runtime parameter. A configuration generator emits combinations of optimizations.

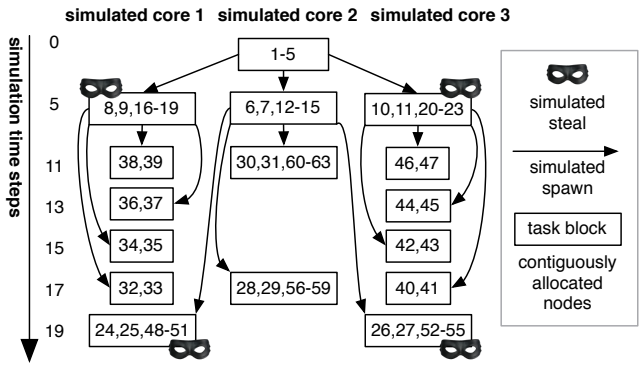


Figure 8: Semi-static scheduling of a blocked 64 node binary tree for 3 cores by partitioning with an approximation of work stealing. Node labels based on a breadth first ordering.

6. Parallel Evaluation

FTL treats blocks as coarse task units and uses a novel approach to precompute their schedule. Our insight is two-fold. First, FTL partitions the blocks among cores for the low-overhead and locality benefits of semi-static scheduling. Second, to approximate the load balancing benefits of runtime work stealing, FTL's novel partitioner *simulates* it. Finally, we autotune over algorithms and settings.

6.1 Following a schedule

The partitioner computes a schedule of blocks that respects the topological dependencies of our traversal patterns. Each core is assigned a distinct sequence of blocks: top-down traversals use a forwards iteration and bottom-up uses backwards. Block layout follows this order. Synchronization is not needed to know which block to evaluate next, per-node overhead is low, and intermediate nodes of blocks are core-local.

Starting evaluation of a block does synchronize. For example, a top-down traversal waits on the parent of a block, and a bottom-up waits on all of its child blocks. The grammar analysis guarantees that these are the only dependencies. The ready check is simple because the owner of the other block is known. E.g., blocks from the same core are guaranteed to be ready.

6.2 Partitioning with approximate work stealing

To partition task blocks of nodes across cores, we introduce a novel heuristic: *approximating* work stealing. The goal is to both load balance and achieve locality.

Work stealing is common for dynamic scheduling [9, 41]. From one time step to the next, a core runs its *most recently* spawned task (under some bound, e.g., 1). This promotes locality within the core. When the local task queue is exhausted, the idle core selects a victim core and picks the task with hopefully the least locality and most remaining work: the *least recently* spawned task.

FTL's partitioner sequentially simulates performing parallel traversal under a work stealing schedule. Instead of actually visiting nodes, it uses a simple estimate of block time as the node count. Figure 8 depicts simulating this process for scheduling 16 blocks on 3 cores. The simulation terminates on the 7th step: it estimates a speedup of 2.13x, which is optimal.

6.3 Tuning

FTL also autotunes its parallel runtime. Options include:

- thread pinning and initial data partitioning
- number of threads, use of hyper-threads, and hardware mapping
- block synchronization: lock variants and yield/stall functions
- use of a thread warm-up period

In addition, FTL autotunes over more established partitioning algorithms and task schedulers beyond the ones above. These include a parallel for-all partitioner, and, using officially recommended TBB patterns [41], several canonical dynamically scheduled divide-and-conquer and graph schedulers. The parallel for-all works by splitting a tree into a sequential root subtree and disjoint descending subtrees that can be computed independently. It avoids synchronization at the expense of load balancing.

7. Evaluation

We evaluate the key contributions of FTL. First, we show that FTL supports parallel language design by automatically finding parallelism in 5 different types of layout languages and by supporting useful queries. Second, we show that autotuning visit order selection yields a 32% speedup. Third, enabling querying and tuning, the set of MSP visit orders for a CSS subset is synthesized in 6 minutes. Fourth, we show that the combination of data layout optimization and abstract work stealing enables strongly scaling parallel evaluation.

Overall, we achieve a 5.2x speedup on a single-socket quad-core processor and a 9.3x speedup on a dual-socket one with 8 total cores. Strong scaling is 4% and 14% below the ideal speedup, respectively. On 4 cores, the `for-all` heuristic coupled with data layout optimizations yields a 1.8x speedup: even without data layout optimizations, the work stealing heuristic increases speedup to 2.8x. Adding data layout optimizations achieves the 5.2x speedups.

7.1 Methodology and Baseline

We evaluate performance on 4 processors: server1 (2 socket x quad-core 2.3GHz AMD Opteron 2356 512KB L2, GCC 4.3 -O3, Linux 2.6.35), server2 (quad-core 2.7GHz Intel Nehalem X5550 8192KB L2,, GCC 4.6 -O3, Linux 2.6.36), laptop (dual-core 2.7GHz Intel Core i7 256KB L2, OS X 10.6.5, GCC 4.5 -O3), and mobile (dual-core 1.6GHz Intel Atom 330 512KB L2, GCC 4.6 -O3, Linux 3.1). Trials are repeated 25 times: 95% confidence of standard error is within 5%. To model the scenario of initial document layout, a single trial consists of first parsing and other preprocessing for a document and then measuring the time to solve layout once.

The grammar is based on a subset of CSS [10, 39]. The grammar includes horizontal, vertical, and text boxes with padding, margin, alignment, sizes, and color options. We test randomly generated 1,000 node trees to reflect the size of modern webpages. One experiment uses a different setup to show the impact of varying the workloads.

Baseline performance is sequential evaluation of the shortest synthesized sequence of parallel traversals. To enable basic optimizations such as hardware prefetching, the data layout is a vector of consecutively laid out (uncompressed) structs. In concurrent work for a similar workload [3], this baseline is reported to exhibit a 1.3x speedup over the random allocation of commercial browsers.

7.2 Designing parallelizable grammars

FTL automatically found parallelism in several layout languages and helped debug them. We tested several types of layout languages specified in α [5] and β [2], which compile to attribute grammars: a flow-based language (a subset of CSS), a grid-based language (e.g., Java Swing [19]), and a proposed flex-box model [16]. We also examined languages beyond documents and boxy UIs: a treemap widget data visualization widget and a concentric circle widget similar to menus found in some smartphone interfaces. FTL found parallelism in each.

FTL's analysis support was used to specify 3 of these systems. We often found ourselves reasonings in terms of sequences of patterns, and, desiring a particular one, would query for it. When it was invalid, we would examine the error output of the synthesize call and then perform more fine-grained data dependency queries. For several grammars, FTL surprised us by finding faster visit orders than those we had in mind. Sometimes, the surprising order revealed an error. For example, we encoded foreign paint calls as passing a phantom value representing canvas state: we missed a dependency, so FTL synthesized parallel paint calls, which would have erroneously overlapped shapes. Simpler than finding the forgotten dependencies is to constrain paint assignments to be in the same sequential traversal.

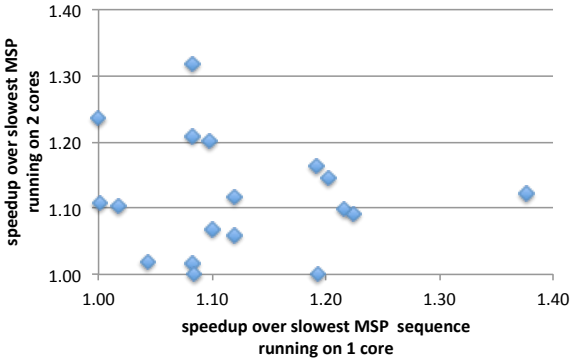


Figure 9: Relative performance of MSP variants.

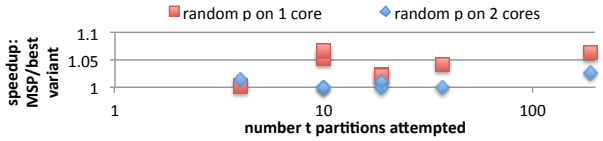


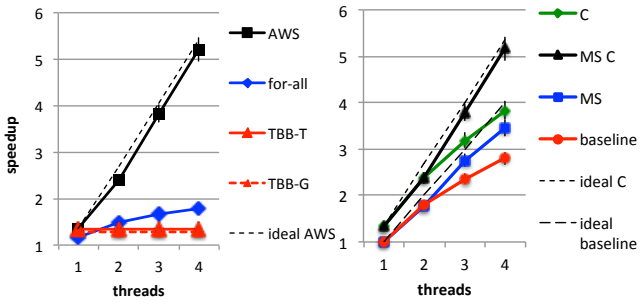
Figure 10: Low benefit of trying non-greedy t for any p in MSP.

7.3 Optimization through synthesis

We evaluate the speedup of tuning over different visit orders (t, p pairs) on laptop, which, in one case, is 32%. We also validate the MSP heuristic, showing no performance loss from ignoring p supersequences and only 0-6% loss from greedily selecting only one t variant.

Varying attribute partitioning t for a particular pattern sequence p improves performance. We generated 24 different p sequences and, for each, anywhere from 2 to 190 different t partitions of A that solve them. As the number of t variants examined increases, the speedup of the fastest vs. slowest variant ($MAX_{a,b \in T} \frac{time(a,p)}{time(b,p)}$) increases. For the p with the most attempted variants ($|T| = 190$), we observe a 12% benefit for sequential evaluation and 18% benefit on dual-core laptop. Generally, the benefit is greater for parallel evaluation than sequential.

Varying p is also beneficial. We generated the MSP set and compare relative performance (Figure 9). For each pair in MSP, we compare its performance to the slowest MSP member when run on single-core (x-axis) and on dual-core (y-axis). Interestingly,



(a) Task scheduler comparison (b) Tuning abstract work stealing

Figure 11: Abstract work stealing is the most effective scheduler (a) and traversal scales due to data layout optimizations (b). Visit order is the shortest sequence of parallel patterns. *Figure (a)*: `AWS` = abstract work stealing, `for-all` = semi-static for-all, `TBB-T` = TBB tasks, `TBB-G` = TBB graph; all with compressed layout. *Figure (b)*: `baseline` = abstract work stealing, `C` = adding pointer compression, `MS` = adding tuning over thread pinning assignment.

there is no overlap in the top 4 visit orders for dual-core vs. single-core evaluation. Using the same traversal visit order across different hardware can be very inefficient.

The MSP heuristic finds visit orders that are better or competitive with related non-MSP variants. Recall a non-MSP variant is a (p, t) pair where p is a supersequence of a pair in MSP or t a permutation. All supersequence p variants cause slowdowns: the overheads of extra traversals seem prohibitive. Next, we check heuristic choice to greedily pick t such that an attribute is computed in the first possible traversal. Examining speedup of trying non-MSP t variants ($MAX_{(u,p)\in \text{non-MSP}\cup\{t\}} \frac{time(t,p)}{time(u,p)}$ shown in Figure 10) reveals a benefit of less than 3%. Furthermore, the speedup is lower for shorter, more optimal visit orders, for which we note there are also fewer t variants.

7.4 Fast synthesis

We evaluated query language performance, finding the synthesizer and query tool fast enough to be part of the edit/compile cycle. The underlying synthesis algorithm is interpreted sequentially in Prolog on laptop.

Queries about dependencies between attributes are interactive. In contrast, streaming enumeration of visit orders ran for a day without terminating. By restricting the search to MSP variants, all evaluators are found and code generated in 6 minutes. Incremental queries or those inspecting individual evaluators, such as requesting all starting with pattern sequence td bu, ran interactively without precomputing the traversals as they prune the search space.

7.5 Effective task scheduling

Scheduling with an abstract work stealing partitioner (aws) is the only scheduler that yielded more than a 1.5x speedup. We also show strong scaling for different hardware, grammars, and trees.

Sequential overhead is low when comparing aws to sequential evaluation: the performance difference is within the error bound. Our intuition is that both schedules are depth-first traversals over blocks, with only minor differences in logic when picking the next block. We do not measure schedule generation, but note that in the best schedule for laptop, maximum block size for a subtree is 102; the number of blocks for the partitioner to traverse is low.

Using aws outperforms any of the other task scheduler tested by at least 3x on one socket of server1 (Figure 11(b)). Dynamic task scheduling in TBB nor its dynamic graph scheduler yielded speedups. Using a different semi-static partitioning, `parallel-for`, yields a 1.5x speedup over its sequential version. Analyzing the expected speedups under our simple cost model, we found the schedule imbalanced. Even without data layout optimizations, our aws achieves a 2.8x speedup. Incorporating them, the total speedup is 5.2x over the sequential baseline.

	Total speedup Cores				% of ideal scaling Cores		
Processor	1	2	4	8	2	4	8
server1	1.4x	2.4x	5.2x	9.3x	89%	96%	86%
server2	1.4x	2.5x	5.2x	n/a	89%	94%	n/a
laptop	1.4x	2.1x	n/a	n/a	78%	n/a	n/a
mobile	1.3x	2.2x	n/a	n/a	86%	n/a	n/a
average	1.4x	2.3x	5.2x	9.3x	86%	95%	86%

Figure 12: Speedups and strong scaling across different hardware. Baseline is uncompressed sequential traversal. Visit order is the shortest sequence of parallel patterns. Right columns depict percent of perfect scaling. For example, 100% for 4 cores would indicate a 4x speedup over evaluation on one core using the same optimizations.

7.6 Data layout optimization and tuning

We show that our data layout optimizations speed up sequential performance by 26% and are an important part of parallelization by improving strong scaling to reach 4% of ideal from an initial low 40% of ideal. Furthermore, we show benefits from autotuning over optimizations and their parameters.

Figure 11(b) depicts the sequential and parallel benefits of 2 data layout optimizations for abstract work stealing. The optimizations are compressing data and/or changing thread-to-core mapping. First, using both layout optimizations improves sequential performance by 1.3x. Second, data layout optimize achieves strong scaling. Using neither optimization, speedup is 30% from ideal scaling on 4 cores (2.8x out of ideal total 4x), but with both, it is 4% from ideal scaling (5.2x out of ideal total 5.4x). Data layout optimization improves sequential performance and achieves strong scaling of parallel evaluation on one socket of server1. Speedup less strongly scales when adding cores on the second speedup: 14% of ideal for a 9.3x total speedup.

Autotuning is effective. For example, block size depends on the grammar, hardware cache size, and number of threads. On server1, block sizes between 15-30 are best across various core counts: on both 4 and 8 cores, for example, they have an average 18% speedup over using a block size of 60. Even picking the number of threads should be tuned. Adding a 5th core to server1, which involves running on a second socket, has only a relative speedup of 7%, with scaling only strong again on adding further second socket cores. Finally, the same algorithms were typically selected but not always. As a sample exception, the choice between depth-first and breadth-first layout within a block only mattered on mobile.

7.7 Varying workloads

Different workloads perform well under our approach. In Figure 12, we show strong speedups across different hardware. Sequential speedup from data layout optimization is 28-40% and scaling is generally 4-14% of ideal. One exception is laptop, which is only 78% of ideal scaling: APIs in OS X does not support our thread pinning optimizations. To model more complex languages, we repeated each loop 5 times: scaling is still strong on server1 (e.g., 86% of ideal on 4 cores). We also modeled increasing page

size by 5x, as might be seen in a data visualization. For example, 4 cores of server1 achieves 83% of ideal scaling.

8. Related Work

Layout language implementation is well-studied. Badros presents the Cassowary [7] linear constraint solver, which Badros et al. use for the SCWM window manager [8]. Wang and Wood [46] show formatting constraints that require richer solvers; Lin [35] implements basic document layout by interleaving Cassowary calls with manually implemented solvers. Due to concerns in performance, expression, etc., commercial webpage layout engines are manually implemented as a sequential sequence of visitors [22, 24].

Parallel browsers are of recent interest. Jones et al. [27] propose both parallelizing individual components and employing a concurrent architecture. Commercial browsers perform GPU-accelerated rendering and, based on subsequent work by Meyerovich and Bodík [39] and Badea et al. [6], layout preprocessing (templating) on multiple cores. Other compute-bound browser components, such as lexing [27] and parsing [20], are examined in literature.

Parallel layout is an open challenge. Brown [11] propose applying task parallelism, which Meyerovich and Bodík [39] implement using Cilk [9] and TBB [41] to weakly scale a subset of CSS. Burckhardt et al. [12] show the task parallel Revisions framework also achieves weak scaling. FTL strongly scales on multicore hardware.

Browsers load independent resources in parallel, which can be used for parallel layout by decomposing a page into independent units. Concurrent work by Anonymous [4] 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 [39] for parallel rendering. We parallelize by optimizing just the layout engine.

Attribute grammars, introduced by Knuth [30] to define language semantics, are tractable for static analysis and optimization. Saraiva and Swierstra [43] specify non-automatic HTML table layout with attribute grammars and Meyerovich and Bodík [39] examine CSS.

Kastens [29] presents a sequential evaluator generator for the class of ordered attribute grammars; most grammar compilers are similar. FTL instead finds an individual variant using a monotonic search and can generate multiple variants. Our approach supports tuning as well as constraints and queries over evaluators. The MSP heuristic refines the usual greedy heuristic to consider multiple p sequences.

Jourdan [28] surveys parallel attribute grammar evaluators. Most similar to FTL is their technique of analyzing the grammar for subtree visit recursions with no data dependencies that can thus be computed independently in parallel. Motivating our work, Jourdan also presents a work stealing scheduler and observes challenges in strong scaling. Our use of specialized parallel patterns is novel.

Incremental evaluators are a complimentary approach [12, 42].

Data layout optimization is well-studied. For compression, we encode a variant of the pointer representation of Lattner and Adve [34]. Our use of a small set of pointers is a common manual practice. More aggressively, browsers trade spatial locality for smaller representation size by aliasing identical node style data. Ananian and Rinard [1] propose compressing individual data fields with a bit flag and auxiliary structures: the flag indicates whether to use a default class value or lookup a non-default instance value.

Fine-grained approaches such as the structure split coallocation of Chilimbi et al. [13] improve spatial locality. We optimize temporal locality by tiling [25], pinning, and tuning the tree layout; concurrent work by Jo and Kulkarni describes a similar technique for DAGs [26]: point blocking. They study large graphs; we find incorporating further optimizations is useful for small ones.

In concurrent work [3], we manually cluster layout to optimize single-core behavior: branch prediction, SIMD utilization, applicability of compiler optimizations such as hoisting, etc. FTL's code generation and autotuning support might automate this technique.

Data structure selection is examined early on by Low [36] for abstract data types in an ALGOL-60 variant. Recent work by Hawkins et al. examines exposing and implementing multiple pointer-based representations [23] satisfying the same relational interface. Supporting data layout optimizations used by FTL remains a challenge.

The superoptimization project [38] by Massalin generates functionally equivalent programs based on a low-level instruction set andFrigo and Johnson [21] generates FFTW patterns. FTL searches for valid different traversal sequences.

FTL is similar to the ATLAS [47] framework for linear algebra in that it autotunes over parameters such as block size to optimize for a particular device. In contrast, cache oblivious algorithms for FFTW [21] asymptotically optimize for general architectures but may be less efficient on any individual device. Autotuning is actively being applied to further domains, such as work in stencils [15] by Datta et al.: we examine computations over trees.

MIMD task and graph languages generalize over grammars. To exploit multicore architectures for task parallel programs, work stealing systems such as Cilk [9] and TBB [41] use a runtime scheduler to load balance tasks. Programmers must provide a task parallel decomposition and manually manage data layout. We were not able to achieve strong scaling. Similar to FTL, Kulkarni et al. [32] specialize work stealing for iterative computations over graphs: blocking subgraphs improves locality and amortizes overheads. We further decrease overheads with more static approaches.

Cluster computing for large graphs as in Pregel [37] has resurgent interest. We focus on the complimentary single-node case.

Kwok and Ahmad [33] survey static DAG scheduling. It is NP-complete when node weights are non-uniform. Unlike FTL, many techniques assume message passing and no temporal locality. We approximate work stealing as a fast, simple, and effective semi-static partitioner. Irrespective of the schedule quality, much of our work is to eliminate overheads.

9. Conclusion

We have presented FTL, a system for designing and implementing parallel layout languages. We show how to achieve strong scaling by combining contemporary and novel techniques, and believe FTL is a powerful foundation for future work.

The small nature of the computations challenge effective parallel evaluation. First, we show the importance of finding a good sequence of tree traversals, which we address using a novel synthesis-based grammar compiler. Using it, we provide an autotuner over schedules and a novel parallel language design tool. Second, we show how to optimize data layout and third, present a new semi-statically task scheduler: without both, parallel evaluation did not scale. Overall, we achieve a 5.2x speedup using 4 cores and even higher speedups on multi-socket devices. Individual results, such as our parallel language analysis tools or abstract work stealer, can be used independently.

FTL is a foundational approach towards building future web browsers and layout languages because it separates layout language design, analysis, and implementation. A key demonstrated result is that we achieve the first strongly scaling multicore layout engine. Further open optimization challenges, such as SIMD evaluation [3], may now also be tractable. Challenges not related to performance might now also be overcome. For example, we are considering verification and debugging support for layout languages.

References

[1] C. Ananian and M. Rinard. Data size optimizations for Java programs. *ACM SIGPLAN Notices*, 38(7):59–68, 2003.

[2] Anonymous. β. 2011.

[3] Anonymous. Single-core data layout optimizations for trees. 2011.

[4] Anonymous. Proxy server optimizations for parallel layout. 2011.

[5] Anonymous. α. 2011.

[6] C. Badea, M. R. Haghighat, A. Nicolau, and A. V. Veidenbaum. Towards parallelizing the layout engine of Firefox. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.

[7] G. J. Badros. *Extending Interactive Graphical Applications with Constraints*. PhD thesis, University of Washington, 2000. Chair-Borning, Alan.

[8] G. J. Badros, J. Nichols, and A. Borning. Scwm: An extensible constraint-enabled window manager. In *USENIX Annual Technical Conference, FREENIX Track*, pages 225–234, 2001.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216, August 1995.

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

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

[12] S. Burckhardt, D. Leijen, C. Sadowski, J. Yi, and T. Ball. Two for the price of one: A model for parallel and incremental computation.

[13] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. *SIGPLAN Not.*, 34:1–12, May 1999.

[14] A. Colmerauer. An introduction to Prolog III. *Commun. ACM*, 33: 69–90, July 1990.

[15] 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 *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[16] N. Deakin, I. Hickson, and D. Hyatt. Flexible box layout module, 2011.

[17] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[18] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th international conference on Supercomputing*, pages 68–75. ACM, 1997.

[19] R. Eckstein, M. Loy, and D. Wood. *Java Swing*. O'Reilly & Associates, Inc., 1998.

[20] S. Fowler and R. Bodík. Parallel JavaScript parsing. 2011.

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

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[23] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Data representation synthesis. In *Proc. of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2011.

[24] D. Hyatt. WebCore rendering III layout basics. `http://www.webkit.org/blog/116/webcore-rendering-iii-layout-basics`, 2007.

[25] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 319–329, New York, NY, USA, 1988. ACM.

[26] Y. Jo and M. Kulkarni. Enhancing locality for recursive traversals of recursive structures. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 463–482. ACM, 2011.

[27] C. Jones, R. Liu, L. Meyerovich, K. Asanović, and Bodík. Parallelizing the web browser. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, pages 7–12. USENIX Association, 2009.

[28] M. Jourdan. A survey of parallel attribute evaluation methods. In *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 234–255. Springer Berlin / Heidelberg, 1991.

[29] U. Kastens. Ordered attributed grammars. *Acta Informatica*, 13:229–256, 1980.

[30] D. E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.

[31] D. E. Knuth. *The TEXbook*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[32] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *SPAA*, pages 217–228, 2008.

[33] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31:406–471, December 1999.

[34] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, Chigago, Illinois, June 2005.

[35] X. Lin. Active layout engine: Algorithms and applications in variable data printing. *Computer-Aided Design*, 38(5):444–456, 2006.

[36] J. Low. Automatic data structure selection: an example and overview. *Communications of the ACM*, 21(5):376–385, 1978.

[37] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data*, pages 135–146. ACM, 2010.

[38] H. Massalin. Superoptimizer: a look at the smallest program. *SIGPLAN Not.*, 22:122–126, October 1987.

[39] L. A. Meyerovich and R. Bodík. Fast and parallel webpage layout. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 711–720, New York, NY, USA, 2010. ACM.

[40] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Trans. Softw. Eng.*, 7:417–426, July 1981.

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

[42] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. In *Symposium on Principles of Programming Languages*, page 35, 1982.

[43] J. a. Saraiva and D. Swierstra. Generating spreadsheet-like tools from strong attribute grammars. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, pages 307–323, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[44] S. Souders. Velocity and the bottom line, July 2009. http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html.

[45] G. Talbot. Confirm a CSS bug in IE 7 (infinite loop). http://bytes.com/topic/html-css/answers/615102-confirm-serious-css-bug-ie-7-infinite-loop, March 2007.

[46] X. Wang and D. Wood. Tabular formatting problems. In *the Third International Workshop on Principles of Document Processing*, pages 171–181. Springer-Verlag, 1996.

[47] 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.