Synthesis and Parallelization for Layout Languages *

Leo A. Meyerovich
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
lmeyerov@eecs.berkeley.edu

Rastislav Bodík (Advisor)
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
bodik@cs.berkeley.edu

Abstract

We examine how to automatically parallelize computations over a tree. In our system, programs are declaratively specified with an extended form of attribute grammars. The key to our approach is a novel synthesizer that statically schedules the program as a composition of parallel tree traversals. We show how synthesis enables new linguistic primitives for automatic parallelization where programmers may specify any part of the schedule and rely upon the synthesizer to complete the rest. Furthermore, by identifying tree traversals, we also successfully optimize code generation for efficient execution on multicore and GPU hardware.

We evaluate our approach on two long-running challenges. The first challenge is for web browsers on low-power mobile devices. We synthesized the first parallel schedule for a large fragment of the CSS document layout language and report a 3X multicore speedup. The second challenge is of interactive visualizations for exploring big data sets. We built several GPU-accelerated visualizations of 100,000+ data points, which is one to two magnitudes more than achieved with mature high-level languages used for the same domain.

1. Introduction

What would it take to build a 1 Watt web browser for your phone or glasses that does not drain the battery? How about to script a visualization for interactively exploring a big data set? To increase application performance by magnitudes, we have been parallelizing the browser. Previously, we designed parallel algorithms for browser bottlenecks in lexing, parsing, templating, and font handling [9, 14]. This paper examines a less understood bottleneck that consumes 10-25% of the compute time [14, 17]: visual layout. We show how to parallelize a layout engine by automatically generating it.

Prior attempts to parallelize high-level layout languages [5, 6, 14] stumbled on two basic problems. First, *implementation complexity*. For example, the CSS webpage layout language specification spans over 500 pages of informal English, and even sequential implementation is challenged by subtle feature interactions and dependencies. Manual parallelization has therefore carried a high developer burden while automatic parallelization has failed to balance flexibility against performance. Second, browsers require *strongly scaling* parallel algorithms. Previous attempts have failed to achieve significant parallel speedups without inflating benchmark workloads because of their runtime overheads.

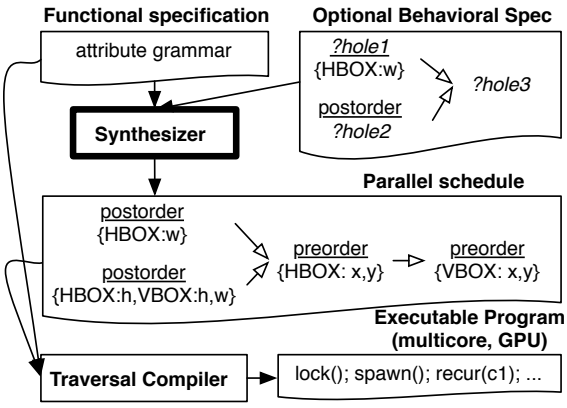


Figure 1: Generating a parallel program with FTL.

We address implementation complexity and strongly scaling parallelism by targeting a more general problem of automatic parallelization. We present the **F**ast **T**ree **L**anguage (FTL) that automatically schedules a program as a composition of parallel traversals over trees. To parallelize layout, we use FTL as a layout engine generator (Figure 1). Given a layout language specified as an attribute grammar [11] and optional schedule constraints, FTL synthesizes a static schedule of parallel tree traversals. The schedule is guaranteed to lay out any document (a tree). Finally, the schedule enables aggressive optimizations in backend code generation.

FTL's use of synthesis enables new constructs for combining manual and automatic parallelization. FTL can automatically schedule, such as by correctly deciding whether a long-running data dependency can be isolated for parallel execution. However, parallelization also needs manual control.

* 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.

```
1  class HBox(paintBorder, fonts) implements Box
2   children child : Array<Box>
3   child[0].x = x
4   w[0] = 0
5   h[0] = 0
6   loop i in child.length:
7    child[i].y = y
8    child[i > 0].x = child[i − 1].x + child[i − 1].w
9    w[i > 0] = w[i − 1] + child[i].w
10   h[i > 0] = max(h[i − 1], child[i].h)
```

Figure 2: Horizontal box widget specified in FTL.

For example, local modifications to a declarative program can induce global serializing changes to the schedule. We introduce behavioral specification constructs for communicating to the compiler and across developers which schedule changes are breaking or non-breaking. For example, the programmer can specify the sequence of traversals to use. The synthesizer would find a correct and efficient ordering of computations across the traversals, or report how the desired schedule conflicts with data dependencies in the program.

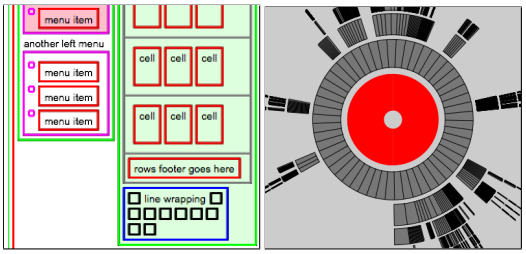
This article explores our schedule synthesizer: its functional specification language, parallel programming with synthesis, and schedule optimization. Finally, we evaluate FTL on two important challenges: multicore browsers for low-power devices and GPU-acceleration of big data visualization. Our synthesizer is further described in the recent proceedings of PPoPP [15] and our GPU-accelerated big data visualization framework is online (`http://sc-lang.com`).

2. Functionally Specifying Layout

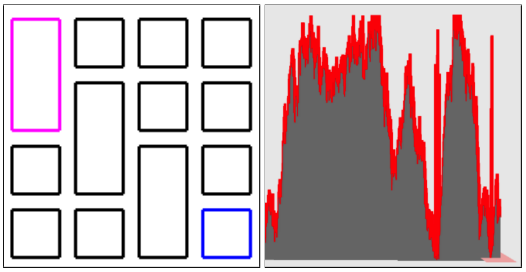
This section shows how to program an automatically parallelizable horizontal box layout engine with FTL. Designing FTL's specification language to support layout programs was non-trivial. For comparison, Cassowary's [2] linear constraints can be optimized, but they cannot express line breaking [13]. Conversely, higher-order attribute grammars [16] are flexible but difficult to optimize.

FTL's functional specification language is similar to common general purpose languages. Consider our specification of a horizontal box in Figure 2: it uses classes, traits, arithmetic, and, as in the max() call, arbitrary pure functions. Supporting features that programmers expect required expressive extensions to classic ordered attribute grammars [11], such as our synthesis of loops as a generalization of uniform recurrence relations [10, 15]. Functional specification in FTL is more relaxed than in imperative and eager languages because statement order is ignored. To control imperative behavior (e.g., parallelization), we introduce an orthogonal and optional scheduling language (Section 3).

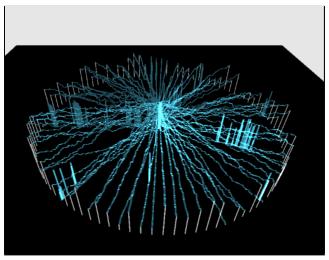
We made automatic scheduling tractable by enforcing two restrictions from attribute grammars [11]. First, node attributes are singly-assigned; each one appears on the left-hand side of only one assignment. Second, we localize heap access. Nodes only access attributes of their neighbors.



(a) Flow-based **(b)** Radial



(c) Grid-based **(d)** 2D Charts



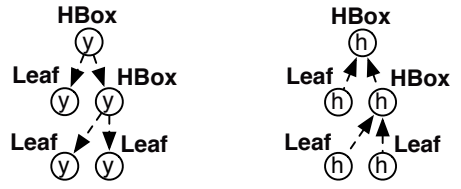
(e) 3D Charts

Figure 3: Parallel layout engines synthesized using FTL.

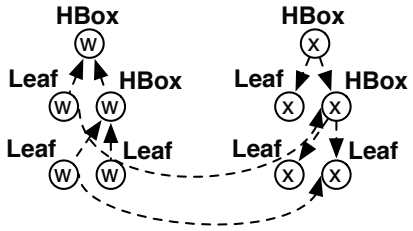
Our combination of language features and restrictions supports important common cases. For example, we automatically generated parallel implementations for flow-based layouts, radial layouts, grids, charts, and even 3D visualizations (Figure 3). Furthermore, the layouts in Figure 3 (b), (d), and (e) are also interactive or animated in real-time, which we achieve by encoding user state and the time as input attributes for the layout computes over.

For the HBox in Figure 2, FTL's synthesizer finds the parallel schedules in Figure 4 (d). For example, the schedule "td{Y}; bu{H,W}; td{X}" specifies a sequence of three top-down (td) and bottom-up (bu) traversals over the document. Both td and bu traversals are parallelizable (parallel preorder and postorder, respectively). The terms inside the braces such as W and H specify what attributes will be computed on a node when the traversal reaches it.

Not all schedules are valid, but the synthesizer finds many that are. For the HBox widget, it found 5 different correct schedules made from different compositions of traversals



(a) Traversal td{Y} **(b)** Traversal bu{H}: bottom-up



(c) Traversals bu{W} ; td{X}: bottom-up then top-down

1) td{Y}; bu{H}; bu{W}; td{X} 2) td{Y}; bu{H,W}; td{X}
3) bu{H,W}; td{Y,X} 4) (td{Y} || bu{H,W}); td{X}
5) (bu{H} || bu{W}); (td{Y} || td{X})

(d) Equivalent traversal sequences made by combining the traversals

Figure 4: Tree traversal sequences that solve any HBox document. Shown on a document instance. Edges denote dynamic data dependencies.

and partitions of attributes to compute in each one. Every scheduled attribute computation is guaranteed to respect the dynamic data dependencies in layout instances. For example, x attributes are computed from w attributes, so x attributes will never be scheduled for an earlier traversal. Figure 4 (c) shows a schedule safely computing the x and w attributes. The synthesizer automates such reasoning.

3. Parallel Programming with Synthesis

We show how FTL uses synthesis to enables new approaches to *guiding* and *debugging* parallelism.

3.1 Interactive Parallelism Debugging

We often used FTL to debug parallelization ideas for complicated programs. For example, we can ask FTL whether widths can be scheduled for an initial parallel traversal:

$$\text{Sched=[bu}(A) \mid \boxed{\textit{REST}}\text{]} \quad \wedge \quad \text{member}(\text{w}, A)$$

The query is written with the schedule syntax of Figure 4 (d) extended with standard Prolog [18] and the *hole* primitive $\boxed{\textit{VAR}}$. It asks the synthesizer to find a schedule Sched that starts with a bottom-up traversal that includes attribute w in its set A of attributes. The synthesizer outputs all such schedules by finding safe solutions for each hole (and Prolog variables such as A). If there are no such schedule completions, the synthesizer helps narrow down the error by reporting the biggest valid schedule up to a violation of the above query.

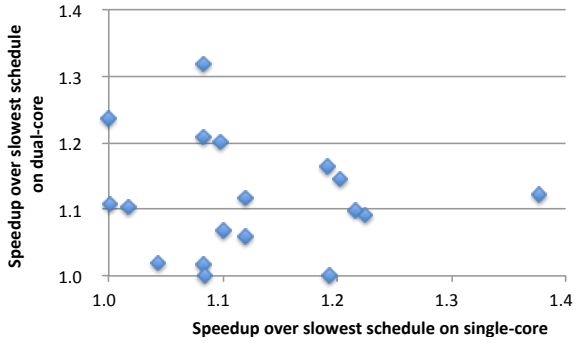


Figure 5: Speedup of different synthesized schedules over the worst synthesized schedule. Each axis denotes different hardware.

3.2 Behavioral Specification of Parallelization

Programmers can *declaratively guide* automatic parallelization using a mechanism similar to the above query language.

For example, FTL can "lock in" the above schedule. We found this to be an important yet lightweight approach for structured parallel programming, such as for preventing program edits from breaking a parallelization scheme. We do this by simply reusing the above query as a behavioral specification input. FTL will output a schedule that follows it. If any future program edits introduce data dependencies that violate the schedule constraints, the synthesizer will automatically catch them as compile-time errors.

Our $\boxed{\textit{HOLE}}$ primitive safely and flexibly mixes automatic and manual parallelization. In practice, programmers leave the entire schedule as a hole during initial development. As the program matures, they specify the composition of traversals, such as in "sched=[bu($\boxed{\textit{V1}}$) ; td($\boxed{\textit{V2}}$)]". The synthesizer is trusted with the tedious management of decisions that are unimportant and often change, such as the attributes for a traversal. It verifies any manual scheduling decisions in the behavioral specification against data dependencies extracted from the functional specification, and automates the rest.

4. Automatic Optimization

FTL's design enables two automatic optimization stages: *scheduling* and *code generation*. We describe them in turn.

4.1 Schedule Autotuning

Hardware varies in the amount of processors and memory, so FTL picks the fastest schedule from all that it finds.

We make enumerating schedules tractable with a dynamic program over successively longer schedule prefixes and by

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

Figure 6: Speedups and strong scaling across different task scheduler implementations and hardware. Baseline is a sequential traversal without data layout optimizations. FTL is our semi-static task scheduler. We run both FTL and TBB with data layout optimizations. 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.

pruning optimizations such as iterative refinement [15]. Our autotuner profiles the enumerated schedules on sample documents and selects the fastest. Unlike traditional cache block size autotuners and Petabrick's [1] algorithm selector, whose parameter space is manually provided, our synthesizer automatically infers the set of valid schedules.

We enumerated valid schedules for an extended HBox program and compared performance on 1 and 2 cores. The relative standard deviation for performance of different schedules (σ/μ) is 8%. The best schedules for 1 and 2 cores are different. Swapping them leads to 20-30% performance degradation, and the difference between the best and worst schedules for the two scenarios are 32% and 42%, respectively (Figure 5). Autotuning schedules improves performance.

4.2 Aggressive Code Generation for Tree Traversals

Our code generators combine a schedule with program statements. We also use them for global and low-level optimizations. For example, our multicore backend performs pointer compression [12], subtree tiling [8], and autotuning. Likewise, we only saw strong scaling after adding a novel traversal task scheduler that optimizes for data locality, load balancing, and low overheads by load-time document partitioning with a work stealing heuristic. Commercial browsers are manually written and perform none of these optimizations.

For a challenging test of our optimizations, we ran a small extension to the HBox language on a random tree with only 1,000 nodes (Figure 6). On a dual-core netbook (Atom 330) with little memory, FTL's optimizations achieve 86% of perfect linear scaling from parallelism with a cumulative 2.2X speedup once sequential optimizations are included. On an octo-core Core i7, FTL achieves 86% of perfect linear scaling with a cumulative speedup of 9.3X.

In contrast, when using our memory optimizations but Intel's TBB work stealing scheduler, we did not see any speedups until 8 cores. Likewise, with our custom runtime scheduler but without our memory optimizations (not depicted), scaling was only 72% of the ideal. Adding memory optimizations nearly doubled performance; aggressive memory and scheduling optimizations achieved effective parallelization.

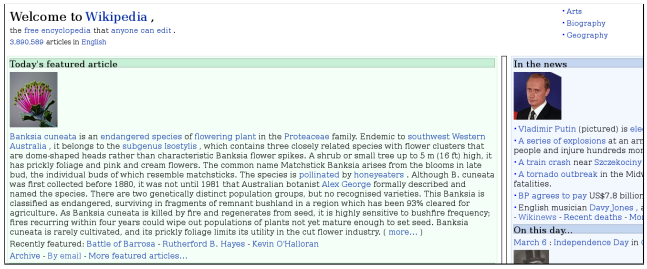


Figure 7: Parallel CSS engine run on Wikipedia.

5. Multicore CSS and GPU Visualization

We evaluate FTL on two important systems. Both challenge FTL's performance and expressiveness.

5.1 Parallelizing CSS

Mobile web browsers are CPU-bound [14] due to power constraints, so we started building the first one to exploit the shift to multicore architectures [9]. A central bottleneck that resisted parallelization was the layout engine. Our solution is to use FTL to create the first mechanized specification of the CSS document layout language and to synthesize the first parallel schedule for a non-trivial fragment of it.

Our CSS specification already handles features such as the box model, nested text, and as depicted in Figure 3 (c), tables. The generated engine successfully renders popular sites such as Wikipedia (Figure 7) with only a few missing features. FTL synthesized a schedule of 9 parallel passes, and we report a 3X multicore speedup on layout for Wikipedia. FTL was sufficiently expressive and efficient thus far.

We used FTL's behavioral specification feature to verify that new layout features did not violate our schedule, and when they did, to try new schedules. As our specification grew, so did our reliance on interactive verification. Our new approach to mixing automatic and manual parallelism helped achieve a result that stumped expert browser developers.

5.2 Big Data Visualization

Visualization tools have not kept pace with increases in data size relative to cloud infrastructure and analytics algorithms. We have therefore been designing a high-level system, SUPERCONDUCTOR, for big data visualization. SUPERCONDUCTOR supports a subset of the domain specific languages (DSLs) in browsers that designers already use for small-scale visualizations. To optimize SUPERCONDUCTOR, we isolated subsets of the DSLs that are tractable for GPU and multicore acceleration.

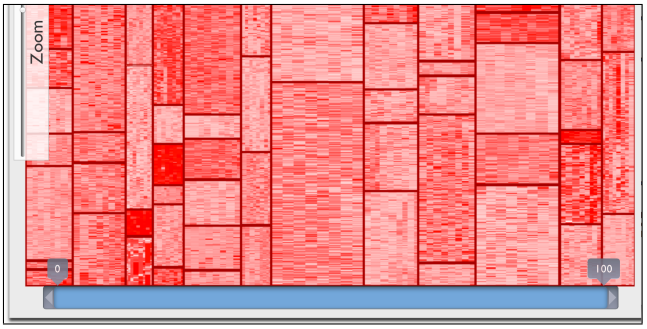


Figure 8: GPU-accelerated treemap of election results.

SUPERCONDUCTOR's sole exception to using web DSLs is for defining custom layouts, for which browsers provide none. We therefore use FTL, and to do so, created a WebCL [7] (GPU) backend that flattens the computation [3] into structure-split level-synchronous breadth-first tree traversals. Furthermore, we optimize for upstream and downstream DSLs. For example, SUPERCONDUCTOR automatically optimizes rendering commands generated by the layout solver by performing vertex memory allocation as a prefix sum and consolidating WebGL rasterization.

Figure 8 shows an interactive visualization of a recent election. SUPERCONDUCTOR visualizes all 96,000 polling stations. In comparison, the popular high-level but sequential D3 library [4] could not visualize more than 2,000 nodes at a frame rate of 27fps. Promisingly, all of our demos use fully automatic parallelization without any manual scheduling.

6. Conclusion

We have shown how to synthesize parallel computations over trees in a way that meets important challenges in mobile web browsing and big data visualization. Furthermore, synthesis enables us to introduce behavioral specification constructs that safely and flexibly mix automatic and manual parallelization. Finally, our design enables two important classes of automatic optimization: schedule autotuning and low-level code generation.

Our approach exposes significant new directions. First, given knowledge of what features a layout uses, a browser might invoke a specialized layout engine that was previously automatically generated. Second, to target machine learning algorithms, we want to generalize synthesis to structured traversals over *graphs* and code generate *distributed* implementations. Finally, our mechanization of the CSS standard may enable building smarter debugging and design tools for the web that further exploit synthesis.

Our outlook is promising – in the meanwhile, we invite readers to try our demos or write their own visualizations at `http://sc-lang.com`.

7. Acknowledgements

Matthew Torok and Eric Atkinson were heroic research assistants for the two 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] G. J. Badros. *Extending Interactive Graphical Applications with Constraints*. PhD thesis, University of Washington, 2000. Chair-Borning, Alan.

[3] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, Apr. 1994.

[4] M. Bostock, V. Ogievetsky, and J. Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, Dec. 2011.

[5] H. Brown. Parallel processing and document layout. *Electron. Publ. Origin. Dissem. Des.*, 1988.

[6] S. Burckhardt, D. Leijen, C. Sadowski, J. Yi, and T. Ball. Two for the price of one: A model for parallel and incremental computation. *OOPSLA*, 2011.

[7] K. Group. WebCL - heterogeneous parallel computing in HTML5 web browsers, 2012. http://www.khronos.org/webcl/.

[8] F. Irigoin and R. Triolet. Supernode partitioning. *POPL*, 1988.

[9] C. G. Jones, R. Liu, L. A. Meyerovich, K. Asanović, and R. Bodík. Parallelizing the web browser. In *HotPar'09*.

[10] R. Karp, R. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 1967.

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

[12] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. *PLDI*, 2005.

[13] X. Lin. Active layout engine: Algorithms and applications in variable data printing. *CAD*, 2006.

[14] L. A. Meyerovich and R. Bodík. Fast and parallel webpage layout. In *WWW 2010*.

[15] L. A. Meyerovich, M. E. Torok, E. Atkinson, and R. Bodik. Parallel schedule synthesis for attribute grammars. In *PPOPP'13*.

[16] J. a. Saraiva and D. Swierstra. Generating spreadsheet-like tools from strong attribute grammars. *GPCE*, 2003.

[17] C. Stockwell. IE8 Performance, August 2008. http://blogs.msdn.com/b/ie/archive/2008/08/26/ie8-performance.aspx.

[18] D. H. D. Warren, L. M. Pereira, and F. Pereira. Prolog - the language and its implementation compared with Lisp. *Artificial intelligence and programming languages*, 1977.