Chapter 6

Optimizing Tree Traversals for MIMD and SIMD

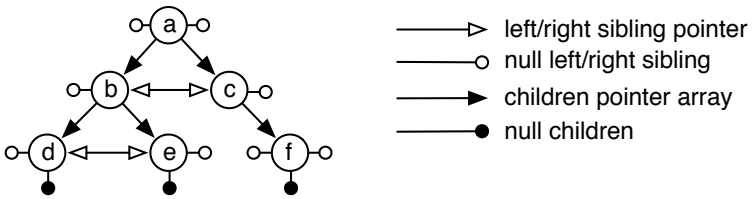
6.1 Overview

For a full language, statically identified parallelization opportunities still require an efficient runtime implementation that exploits them. In this section, we show how to exploit the logical concurrency identified within a tree traversal to optimize for the architectural properties of two types of hardware platforms: MIMD (e.g., multicore) and SIMD (e.g., sub-word SIMD and GPU) hardware. For both types of platforms, we optimize the schedule within a traversal and the data representation. We innovate upon known techniques in two ways:

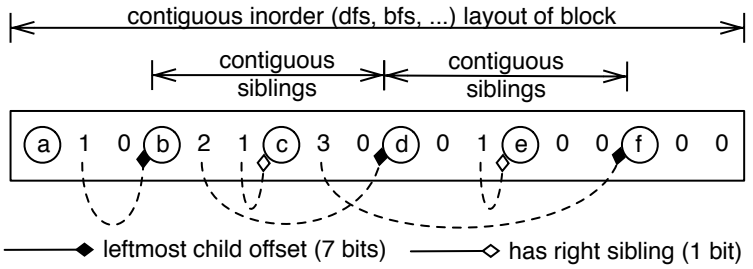
1. **Semi-static work stealing for MIMD:** MIMD traversals should be optimized for low overheads, load balancing, and locality. Existing techniques such as work stealing provide spatial locality and, with tiling, low overheads. However, dynamic load balancing within a traversal leads to poor temporal locality across traversals. The processor a node is assigned to in one traversal may not be the same one in a subsequent traversal, and as the number of processors increases, the probability of assigning to a different one increases. Our solution dynamically load balances one traversal and, due to similarities across traversals, successfully reuses it.

2. **Clustering traversals for SIMD:** SIMD evaluation is sensitive to divergence across parallel tasks in instruction selection. Visits to different types of tree nodes yield different instruction streams, so naıve vectorization fails for webpages due to their visual variety. Our insight is that similar nodes can be semi-statically identified. Thus *clustered* nodes will be grouped in the data representation and run in SIMD at runtime.

Our techniques are important and general. They overcame bottlenecks preventing seeing any speedup from parallel evaluation for webpage layout and data visualization. Notably, they are generic to computations over trees, not just layout. An important question going forward is how to combine them as, in principle, they are complementary.



(a) Naïve pointer-based tree representation



(b) Compressed tree encoding

Figure 6.1: Two representations of the same tree: naıve pointer-based and optimized. The optimized version employs packing, breadth-first layout, and pointer compression via relative indexing.

6.2 MIMD: Semi-static work stealing

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

Data representation: Tuned and Compressed Tiles

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

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

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

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

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

The individual optimizations target several objectives:

- **Compression** Compressing the tree better utilizes memory bandwidth and decreases the working set size. We use two basic techniques: structure packing and pointer compression. Packing combines several fields in the same word of memory, such as storing 32 boolean attributes in one 32bit integer field. Similar to **compression** [**compression**], compression encodes node references as relative offsets (16–20bits) rather than 32bit of 64bit pointers. Likewise, as there are typically few siblings, instead of a counter of number of children (or siblings), we use an isLastSibling bit. Figure 6.1 depicts a tree using pointers and one of our representations: in the example, the compressed form uses 96% fewer bits on a 64-bit architecture.
- **Temporal and Spatial Locality** The above compression optimizations improve locality by decreasing the distance between data. To further improve locality, we support rearranging the data in several ways .

  Tiling [**tiling**] cuts the tree into subtrees and collocates nodes of the same subtree. It improves spatial locality because a node only reads and writes to its neighbors. Likewise, we support breadth-first and depth-first node orderings within a subtree (and across subtrees). Such a representation matches the tree traversal order [**Chilimbi:19999**] and therefore improves temporal locality.
- **Prefetching** We supports several options for prefetching to avoid waiting on data reads. First, the data access patterns with the data layout, so hardware prefetchers might automatically predict and prefetch data. Second, our compiler can automatically insert explicit prefetch instructions as part of the traversal. Finally, runahead processing [**runaheadprocessing**] pre-executes data access instructions. A helper thread traverses a subtree ahead of a corresponding evaluator thread, requesting node data while the evaluator is still computing an earlier thread. We only saw benefits of the first in practice, but leave the others as tunable.

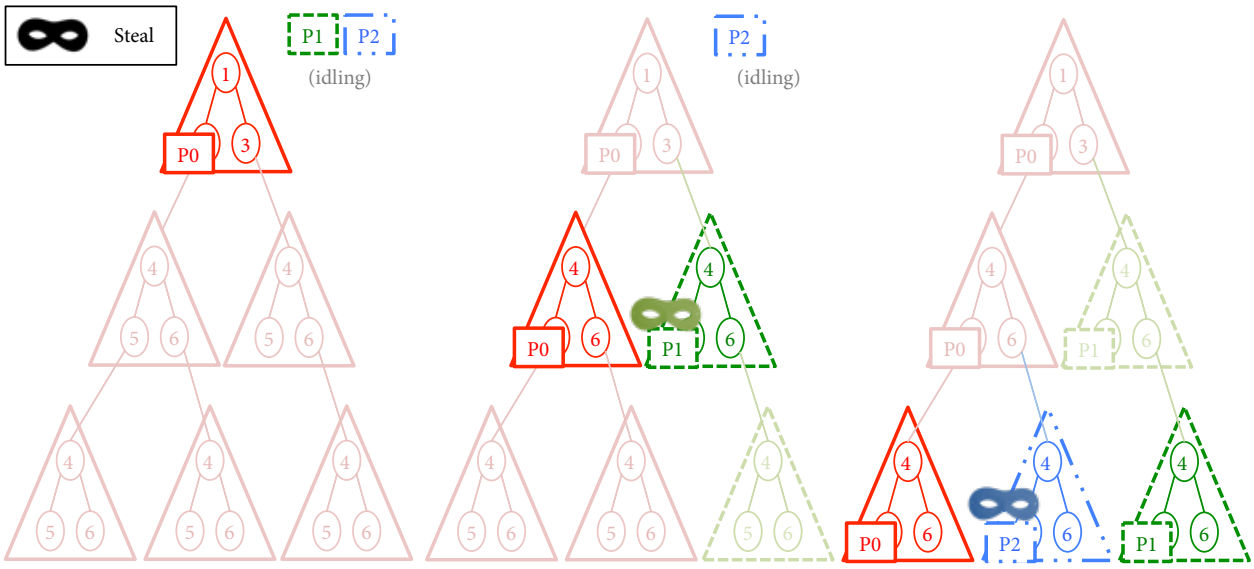


Figure 6.2: **Simulation of work stealing.** Top-down simulated tree traversal of a tiled tree by three processors in three steps.

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

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

Figure 6.1 depicts several of the data representation optimizations: packing, pointer compression, and a breadth-first layout.

Scheduling: Semi-static Work Stealing

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

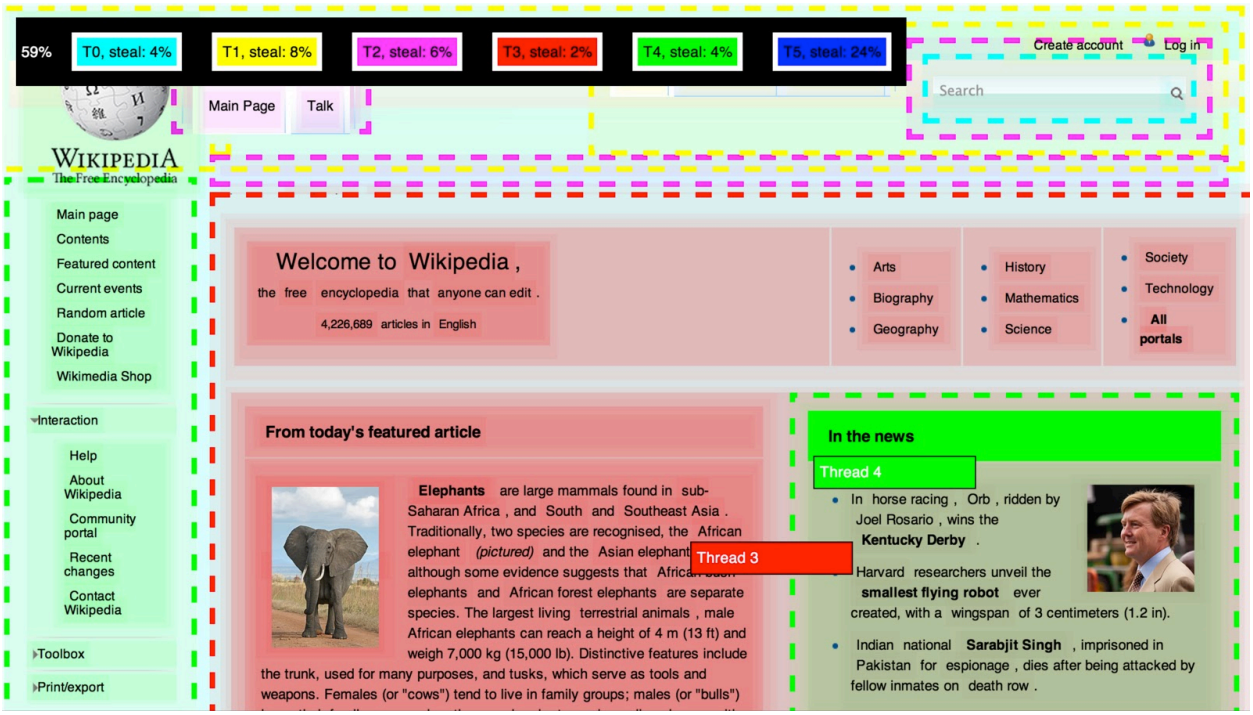


Figure 6.3: **Simulation of work stealing on Wikipedia.** Colors depict claiming processor and dotted boundaries indict subtree steals. Top-left boxes measure hit rate for individual processor.

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

Our algorithm schedules the first traversal using work stealing [[CITE]]. Work stealing was introduced as a dynamic scheduling algorithm that provides load balancing and spatial locality. Figure 6.2 depicts a trace of three processors performing work stealing. Each processor operates on an internal task queue, and whenever a processor exhausts its internal queue, it will *steal* from another processor's queue. In the case of a top-down tree traversal, acting upon an internal queue corresponds to a depth-first traversal of a subtree, and stealing corresponds to transferring ownership of an untraversed subtree. We lower overheads on the first traversal in two ways: we perform task coarsening by scheduling tiles rather than individual nodes, and we simulate the work stealing in one thread on a localized copy of the tiling meta data. The colors of Figure 6.3 show how different processors claim different nodes of a webpage during a parallel traversal: the localization of colors demonstrates the spatial locality of work stealing. Likewise, figure demonstrates that there are relatively few scheduling overheads (steals are indicated by dotted borders).

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

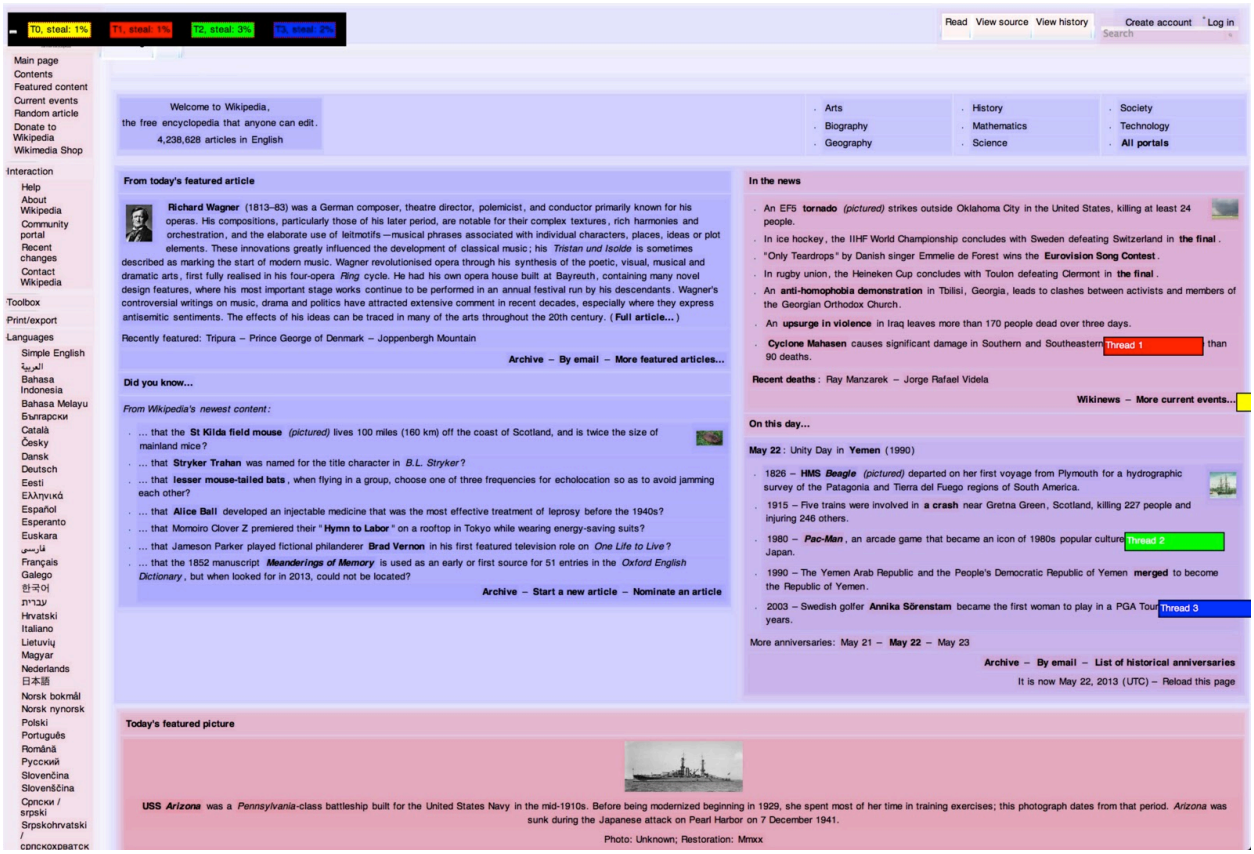


Figure 6.4: **Temporal cache misses for simulated work stealing over multiple traversals.** Simulation of 4 threads on Wikipedia. Blue shade represents a hit and red a miss. 67% of the nodes were misses. Top-left boxes measure hit rate for individual processor.

the overhead, we simulated work stealing for 6 processors on Wikipedia. Assuming uniform compute time per node, 5% of the nodes would trigger stealing. This cost is in addition to constant overhead to processing the internal per-processor task queues. The issue with temporal locality is that a node will be assigned to different processors across multiple traversals. Figure 6.4 shows which nodes must move across processors in a simulation 4 processors performing a sequence of two traversals. 67% of the nodes are red, indicating substantial movement. Both the steal rate and temporal miss rate worsen as the number of processors increase.

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

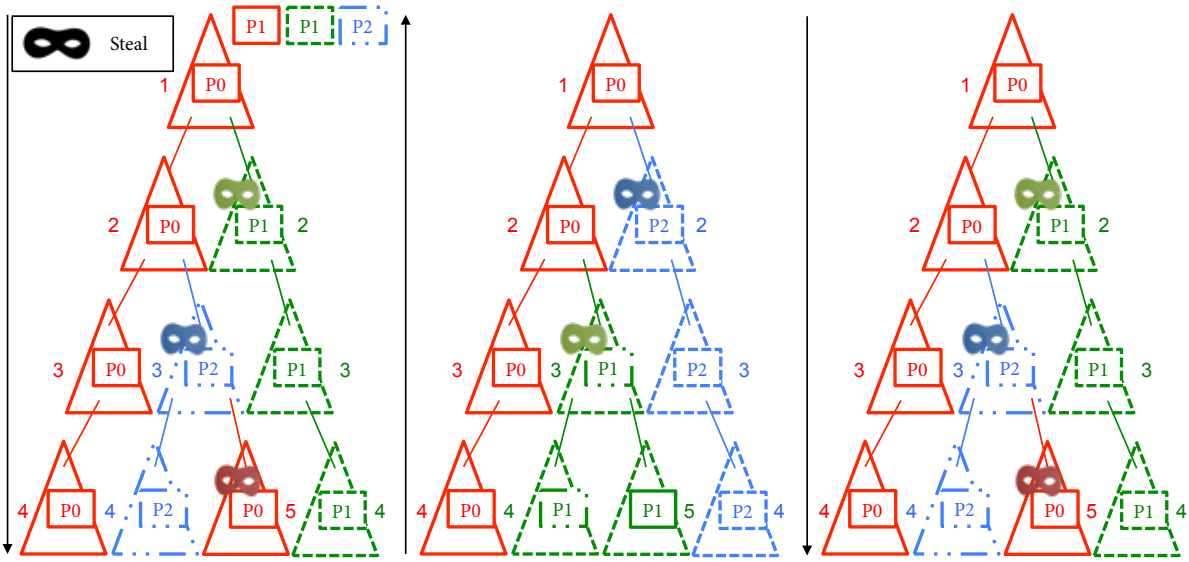


Figure 6.5: **Dynamic work stealing for three traversals.** Tiles are claimed by different processors in different traversals.

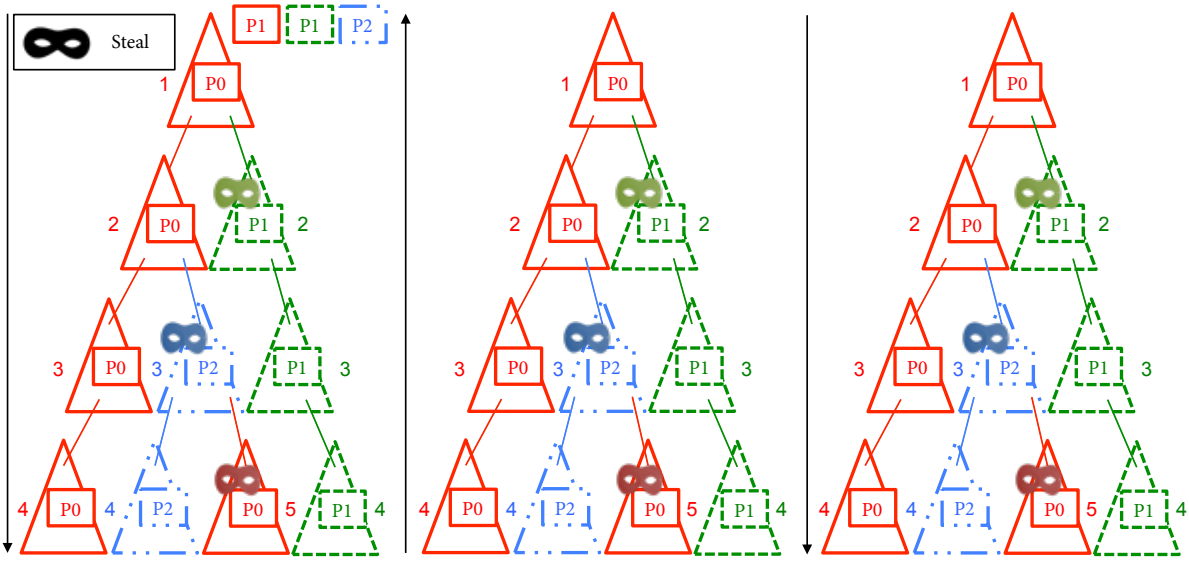


Figure 6.6: **Semi-static work stealing.** Dynamic schedule for first traversal is reused for subsequent ones.

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

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

Evaluation

6.3 SIMD Background: Level-Synchronous Breadth-First Tree Traversal

The common baseline for our two SIMD optimizations is to implement parallel preorder and postorder tree traversals as level-synchronous breadth-first parallel tree traversals. Reps first suggested such an approach to parallel attribute grammar evaluation [[CITE]], but did not implement it. Performance bottlenecks led to us deviate from the core representation used by more recent data parallel languages such as NESL [[CITE]] and Data Parallel Haskell [[CITE]]. We discuss our two innovations in the next subsections, but first overview the baseline technique established by existing work.

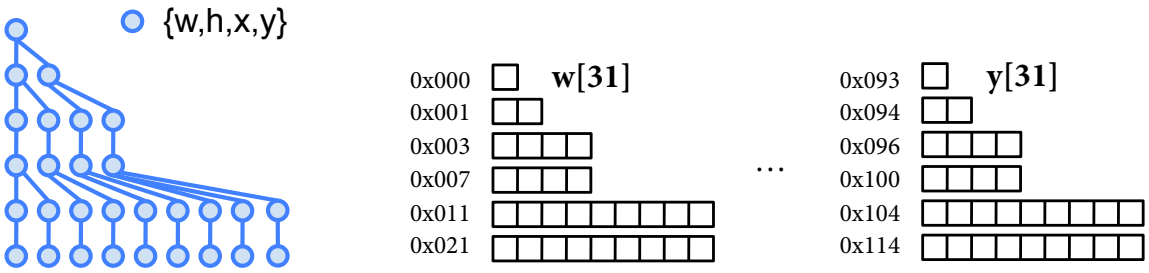
The naıve tree traversal schedule is to sequentially iterate one level of the tree at a time and traverse the nodes of a level in parallel. A parallel preorder traversal starts on the root node's level and then proceeds downwards, while a postorder traversal starts on the tree fringe and moves upwards (Figure 6.7 6.7a). Our MIMD implementation, in contrast, allows one processor to compute on a different tree level than another active processor. In data visualizations, we empirically observed that most of the nodes on a level will dispatch to the same layout instructions, so our naıve traversal schedule avoids instruction divergence.

The level-synchronous traversal pattern eliminates many divergent memory accesses by using a corresponding data representation. Adjacent nodes in the schedule are collocated in in memory. Furthermore, individual node attributes are stored in *column* order through a array-of-structure to structure-of-array conversion. The conversion collocates individual attributes, such as the width attribute of one node being stored next to the width attribute of the node's sibling (Figure 6.7c). The index of a node in a breadth-first traversal of the tree is used to perform a lookup in any of the attribute arrays. The benefit this encoding is that, during SIMD layout of several adjacent nodes, reads and writes are coalesced into bulk reads and writes. For example, if a layout pass adds a node's padding to its width, several contiguous paddings and several contiguous widths will be read, and the sum will be stored with a contiguous write. Such optimizations are crucial because the penalty of non-coalesced access is high and, for layout, relatively few computations occur between the reads and writes.

Full implementation of the data representation poses several subtleties.

```
void parPre(void (*visit)(Prod &), List<List<Prod>> &levels) {
  for (List<Prod> level in levels)
        parallel_for (Prod p in level)
              visit(p)
}
void parPost(void (*visit)(Prod &), List<List<Prod>> &levels) {
  for (Array<Prod> level in levels.reverse())
        parallel_for (Prod p in level)
              visit(p)
}
```

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



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

Figure 6.7: **SIMD tree traversal as level-synchronous breadth-first iteration with corresponding structure-split data representation.**

- **Level representation.** To eliminate traversal overhead, a summary provides the index of the first and last node on each level of a tree. Such a summary provides data range information for launching the parallel kernels that evaluate the nodes of a level as well as the information for how to proceed to the next level.

- **Edge representation.** A node may need multiple named lists of children, such as an HTML table with a header, footer, and an arbitrary number of rows. We encode the table's edges as 3 global arrays of offsets: header, footer, and first-row. To support iterating across rows, we also introduce a 4th array to encode whether a node is the last sibling. Thus, any named edge introduces a global array for the offset of the pointed-to node, and for iteration, a shared global array reporting whether a node at a particular index is the end of a list.

- **Memory compression.** Allocating an array the size of the tree for every type of node attribute wastes memory. We instead statically compute the maximum number of attributes required for any type of node, allocate an array for each one, and map the attributes of different types of nodes into different arrays. For example, in a language of HBox nodes as Circle nodes who have attributes 'r' and 'angle', 4 arrays will be allocated. The HBox requires an array for each of the attributes 'w', 'h', 'x', and 'y' while the Circle nodes only require two arrays. Each node has one type, and if that that

type is HBox, the node's entry in the first array will contain the 'w' attribute. If the node has type Circle, the node's entry in the first entry will contain the 'r' attribute.

- **Tiling.** Local structural mutations to a tree such as adding or removing nodes should not force global modifications. As most SIMD hardware has limited vector lengths (e.g., 32 elements wide), we split our representation into blocks. Adding nodes may require allocation of a new block and reorganization of the old and new block. Likewise, after successive additions or deletions, the overall structure may need to be compacted. Such techniques are standard for file systems, garbage collectors, and databases.

In summary, our basic SIMD tree traversal schedule and data representation descend from the approach of NESL [[CITE]] and Data Parallel Haskell [[CITE]]. Previous work shows how to generically convert a tree of structures into a structure of arrays. Those approaches do not support statically unbounded nesting depth (i.e., tree depth), but our system supports arbitrary tree depth because our transformation is not as generic.

A key property of all of our systems, however, is that the structure of the tree is fixed prior to the traversals. In contrast, for example, parallel breadth-first traversals of graphs will dynamically find a minimum spanning tree [[CITE]]. Such dynamic alternatives incur unnecessary overheads when performing a sequence of traversals and sacrifice memory coalescing opportunities. Layout is often a repetitive process, whether due to multiple tree traversals for one invocation or an animation incurring multiple invocations, so costs in creating an optimized data representation and schedule are worth paying.

6.4 Input-dependent Clustering for SIMD Evaluation

Once the tree is available, we automatically optimize the schedule for traversing a tree level in a way that avoids instruction divergence. Our insight is that we can cluster tasks (nodes) based on node attributes that influence control flow. We match the data layout to the new schedule, and optimize the clustering process to prevent the planning overhead to outweigh its benefit. The overall optimization can be thought of an extension to loop unswitching where the predicate is input-dependent and a sorting prepass guarantees that subintervals will branch identically.

The Problem

The problem we address stems from layout being a computation where the instructions for each node are heavily input dependent. The intuition can be seen in contrasting the visual appearance of a webpage vs. a data visualization. Different parts of a webpage look quite different from one another, which suggests sensitivity to values in the input tree, while a visualization looks self-similar and thus does not use widely different instructions for different nodes. For an example of divergence, an HBox's width is the sum of its children

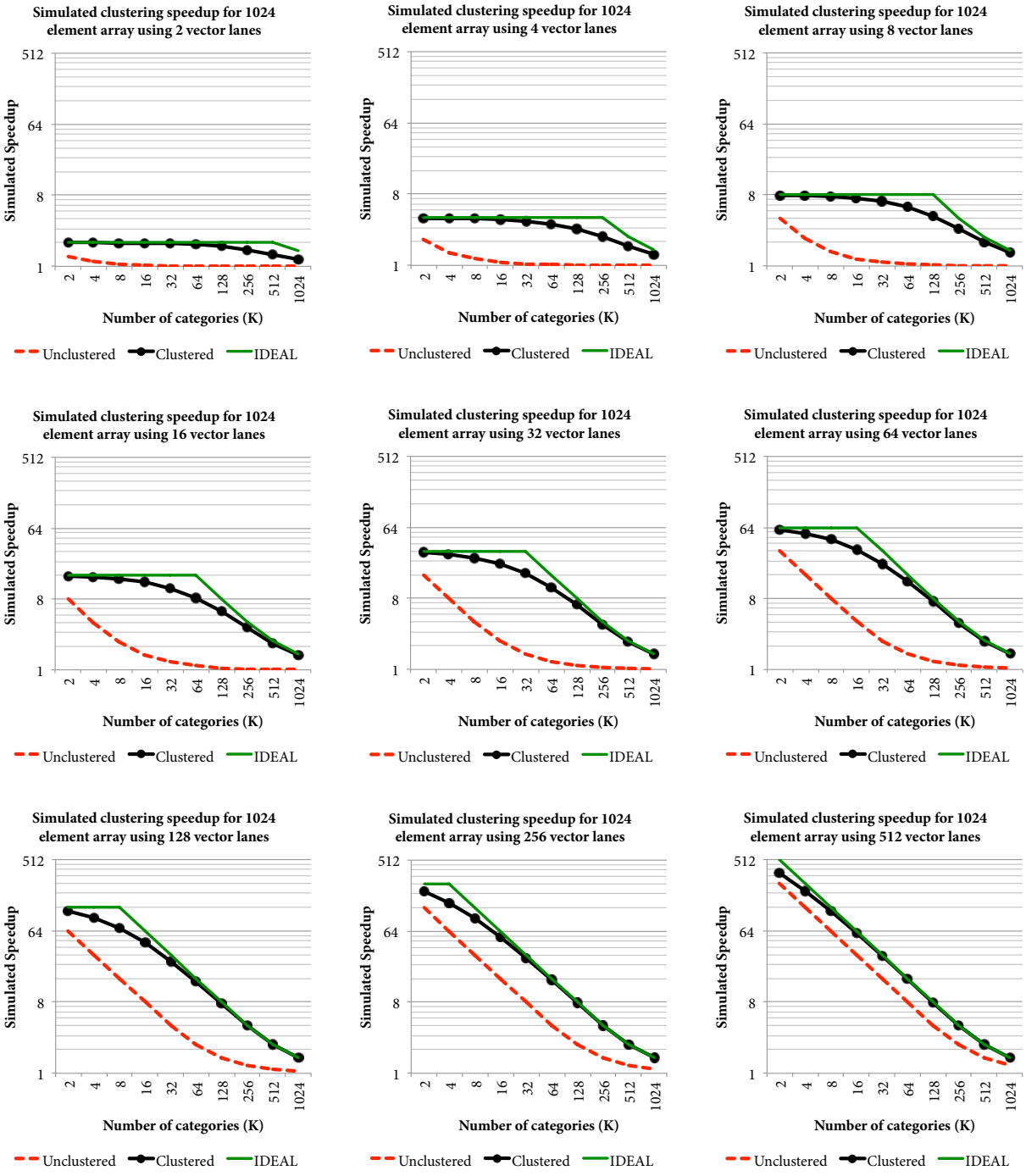


Figure 6.8: **blah**

```
void parPreClustered(void (*visit)(Prod &), List<List<Array<Prod>>> &levels) {
  for (List<Prod> level in levels)
        for (Array<Prod> cluster in level)
              parallel_for (Prod p in cluster)
                    visit(p)
}
```

Figure 6.9: **ASDF.**

widths, while a VBox's is their maximum. The visit to a node (Figure **??**) will diverge in instruction selection based on the node type.

We ran a simulation to measure the performance cost of the divergence. Assuming a uniform distribution of types of nodes in a level, as the number of types of nodes go up (K), the probability that all of the nodes in a group share the same instructions drops exponentially. Figure 6.8 shows the simulated speedup for SIMD evaluation over a tree level of 1024 nodes on computer architectures with varying SIMD lengths. The x axis of each chart represents the number of types and the y axis is the speedup. As the number of choices increase, the benefit of the naıve breadth-first schedule (red line) decreases. It is far from the ideal speedup, which we estimated as a function of the SIMD length of the architecture (maximal parallel speedup, contributing the horizontal portion of the green lines) and the expected number of different categories (mandatory divergences, contributing the diagonal portion).

Code Clustering

Our solution is to cluster nodes of a level based on the values of attributes that influence the flow of control. SIMD evaluation of the nodes in a cluster will be free of instruction divergence. Furthermore, by changing the data representation to match the clustered schedule, memory accesses will also be coalesced. We first focus on applying the clustering transformation to the code.

Figure 6.9 shows the clustered evaluation variant of the MIMD $parPre$ traversal of Figure **??**. The traversal schedule is different because the order is based on the clustering rather than breadth-first index. Changing the order is safe because the original loop was parallel with no dependencies between elements. Computing over clusters guarantees that all calls to a visit dispatch function in the parallel inner loop (e.g., of $visit1$) will branch to the same switch statement case. This modified schedule avoids instruction divergence.

Our loop transformation can be understood as a use of loop unswitching, which is a common transformation for improving parallelization. Loop unswitching lifts a conditional out of a loop by duplicating the loop inside of both cases of the conditional. Clustering establishes the invariant of being able to inspect the first item of a collection sufficing for performing unswitching for a loop over all of the items. Figure 6.10 separates our transformation of $visit1$ (Figure **??**) into using the same exemplar for the dispatch and then loop unswitching.

```
Prod firstProd = cluster[0]
parallel_for (prod in Cluster) {
  switch (firstProd.type) {
    case S → HBOX: break;
    case HBOX → ϵ:
      HBOX.w = input(); HBOX.h = input(); break;
    case HBOX → HBOX1 HBOX2:
      HBOX0.w = HBOX1.w + HBOX2.w;
      HBOX0.h = MAX(HBOX1.h, HBOX2.h);
      break;
  }
}
```

(a) **Clustered dispatch.**

```
Prod firstProd = cluster[0]
  switch (firstProd.type) {
    case S → HBOX: break;
    case HBOX → ϵ:
      parallel_for (prod in Cluster) {
          HBOX.w = input(); HBOX.h = input();
      }
      break;
    case HBOX → HBOX1 HBOX2:
      parallel_for (prod in Cluster) {
        HBOX0.w = HBOX1.w + HBOX2.w;
        HBOX0.h = MAX(HBOX1.h, HBOX2.h);
      }
      break;
  }
}
```

(b) **Unswitched dispatch**.

Figure 6.10: **Loop transformations to exploit clustering for vectorization.**

Clustering is with respect to input attributes that influence control flow, which may be more than the node type. For example, in our parallelization of the C3 layout engine, we found that the engine author combined the logic of multiple box types into one visit function because the variants shared a lot of code. He instead used multiple node flags to guide instruction selection. Both the node type and various other node attributes influenced control flow, and therefore our clustering condition was on whether they were all equal. Using all of the attributes led to too granular of a clustering condition, so we manually tuned the choice of attributes.

Data Clustering

The data representation should be modified to match the clustering order. The benefit is coalesced memory accesses, but overhead costs in performing the clustering should be considered.

Our algorithm matches the data representation order to the schedule by placing nodes of a cluster into the same contiguous array. Parallel reads and are coalesced, such as the inspection of the node type for the visit dispatch. Parallel writes are likewise coalesced.

Reordering data is expensive as all of the data is moved. In the case of our data visualization system, we can avoid the cost because the data is preprocessed on our server. For webpage layout, the client performs clustering, which we optimize enough such that the cost is outweighed by the subsequent performance improvements.

We optimize reordering with a simple parallel two-pass technique. The first pass traverses each level in parallel to compute the cluster for each node and tabulate the cluster sizes for each tree level. The second pass again traverses each level in parallel, and as each node is traversed, copies it into the next free slot of the appropriate cluster. Even finer-grained

parallelization is possible, but this algorithm was sufficient for lowering reordering costs enough to be amortized.

Nested Clustering

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

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

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

6.5 Evaluation

MIMD Data Representation and Scheduling Optimizations

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

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

Table 6.1: **Speedups and strong scaling across different backends (Back) and hardware**. Baseline is a sequential traversal with no data layout optimizations. FTL is our multicore tree traversal library. Left columns show total speedup (including data layout optimizations by our code generator) and right columns show just parallel speedup. Server = Opteron 2356, laptop = Intel Core i7, mobile = Atom 330.

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

Table 6.2: **Parallel CSS layout engine**. Run on a 2356 Opteron.

We first evaluate the perform of our task scheduler (FTL in Table 6.1). Our comparison point is Intel's TBB [**inteltbb**] dynamic task scheduler that performs work stealing [**cilk**], which was the most efficient third-party work stealing library that we tried. We included our data layout optimizations in all calculations because, without them, we saw no speedup. TBB causes slowdowns until achieving no cost (nor benefit) at 8 cores. Our insight is that it suffered from high overheads: switching to scheduling tiles by using our optimized data representation improved performance. Our semi-static working stealing scheduler, however, achieved a 6.9X speedup on 8 cores. We did not see significant further speedups for higher core counts, and hypothesize that it is due to the socket jump. We experimented with other schedulers, such as a simple for-loop over tiles near the fringe of the tree, but the achieved 2X speedup is much lower than the 6.9X of our semi-static work stealer.

Data representation was key to achieving parallel speedups. It achieved 1.2X-1.4X speedups for sequential processing (Table 6.1). However, on 4 cores, it improved performance from 2.8X without data representation optimizations to 5.2X when using them. The difference is 1.9X: our data representation optimizations both complement and improve scheduling optimizations. Without them, parallel performance was poor.

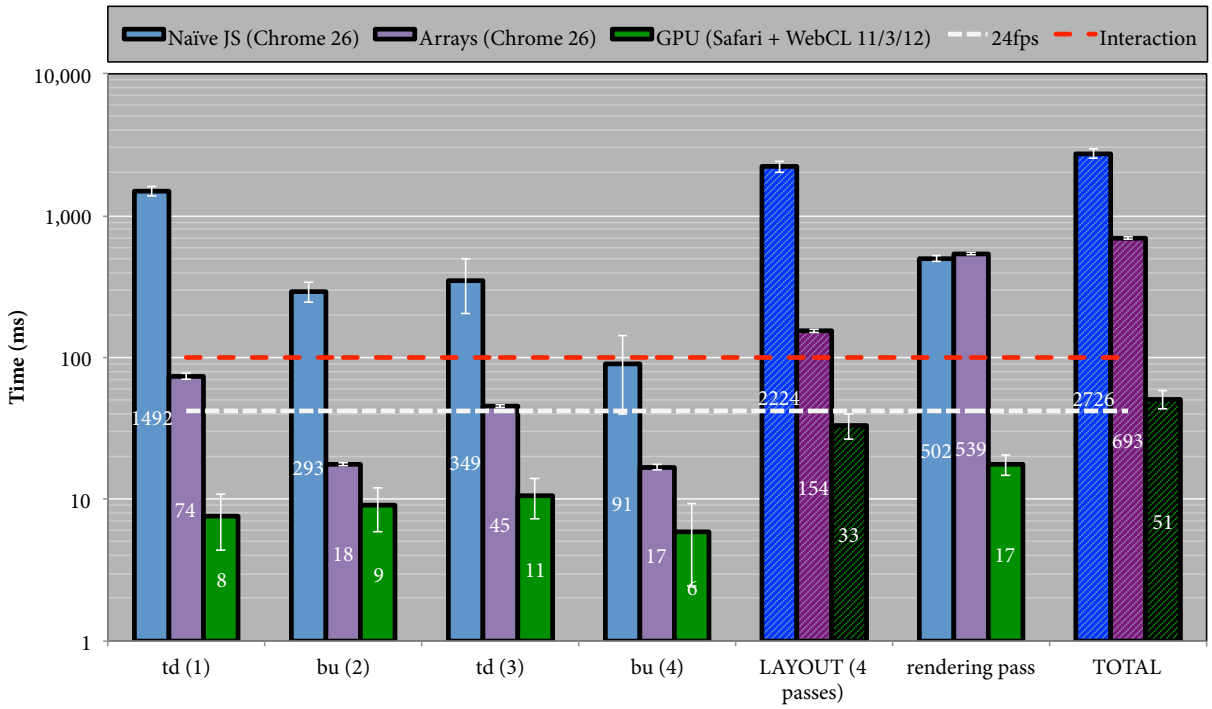


Figure 6.11: **Sequential and Parallel Benefits of Breadth-First Layout and Staged Allocation.** Allocation is merged into the 4th stage and buffer indexing and tessellation form the rendering pass. JavaScript variants use HTML5 canvas drawing primitives while WebCL does not include WebGL painting time (< 5ms). Thin vertical bars indicate standard deviation and horizontal bars show deadlines for animation and hand-eye interaction.

Table 6.2 shows the parallel speedup on running our 9 pass layout engine for two popular web pages that render faithfully with it: Wikipedia and the XKCD blog. Note that the benchmarks do *not* include sequential speedups. The best performance of TBB was a 1.8X speedup on 4 cores, and its speedup on 8 cores was 1.2X. In comparison, our scheduler achieved 2.8X on 2 cores and 3.2X on 8X. Our insight as to why we did not see further benefits is overheads. Across our benchmarks, we generally saw speedups when sequential traversals took longer than a certain amount, but because so many traversals are used for CSS, enough of them are small enough that we do not expect strong scaling. Our intuition is that either a full layout engine is complicated enough that the sequential cost of each traversal will be higher than in our prototype, or even more aggressive data representation optimizations should be performed. As is, we have demonstrated significant 3X+ speedups on real workloads from just the parallelization.

Baseline SIMD Speedups (GPU)

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

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

We first examine the significant sequential benefits.The first 4 groups of columns in Figure 6.11 shows the average time spent on different layout passes and the 6th on the pass for buffer index computation and tessellation. Performing compiler optimizations enables a 14X sequential speedup on layout in the Chrome web browser. No speedup is observed in the rendering pass because the time is dominated by HTML5 canvas calls. We hypothesize part of the sequential benefit is related to our clustering optimization: all of the nodes in a level have the same type, so implicit optimizations such as branch prediction should perform better. Finally, we note that while sequential layout time is a magnitude too slow for real-time animation, our prototype is within 54ms for real-time interaction (ignoring rendering).

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

To better understand the benefit of parallelization, we compared running the layout traversals using multicore vs. GPU acceleration (Figure 6.12) for an early prototype of the layout traversals. Both use breadth-first traversals compiled with OpenCL, except differ on the hardware target. We see that a server-grade multiprocessor (32-core AMD Opteron 2356) can outperform a laptop GPU, but the comparison is unfair in terms of power consumption. TODO compare power ratings.

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

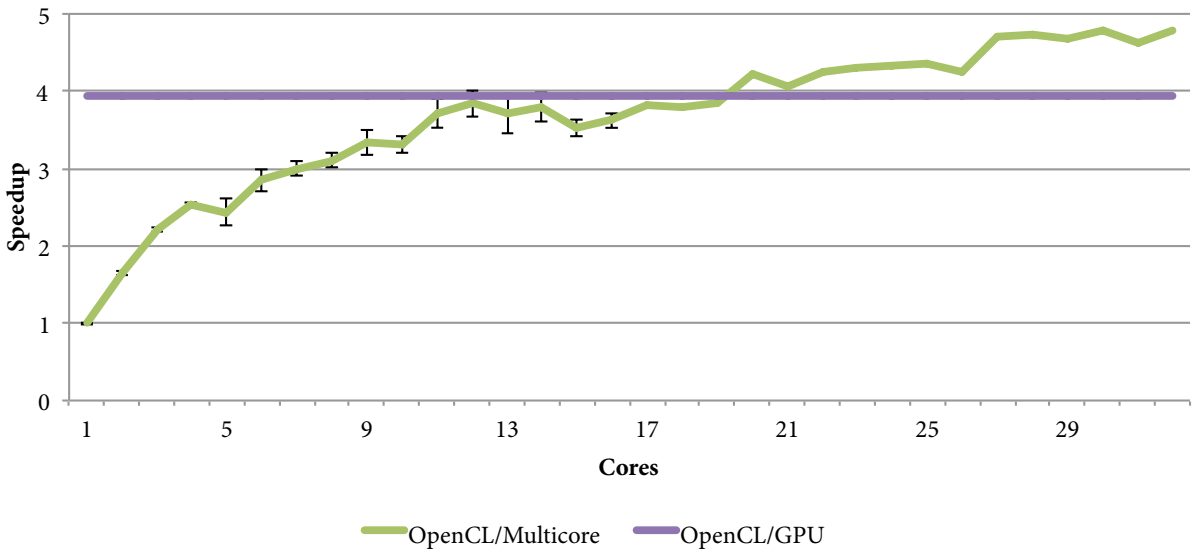


Figure 6.12: **Multicore vs. GPU Acceleration of Layout.** Benchmark on an early version of the treemap visualization and does not include rendering pass.

SIMD Clustering

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

Applicability

We examined idealized speedup for several workloads:

- **Synthetic.** For a controlled synthetic benchmark, we simulated the effect of increasing number of clusters on speedup for various SIMD architectures. Our simulation assumes perfect speedups for SIMD evaluation of nodes run together on a SIMD unit. The ideal speedup is a function of the minimum of the SIMD unit's length (for longer clusters, multiple SIMD invocations are mandatory) and the number of clusters (at least one SIMD step is necessary for each cluster). Figure 6.8 shows, for architectures of different vector length, that the simulated speedup from clustering (solid black line with circles) is close to the ideal speedup (solid green line).

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

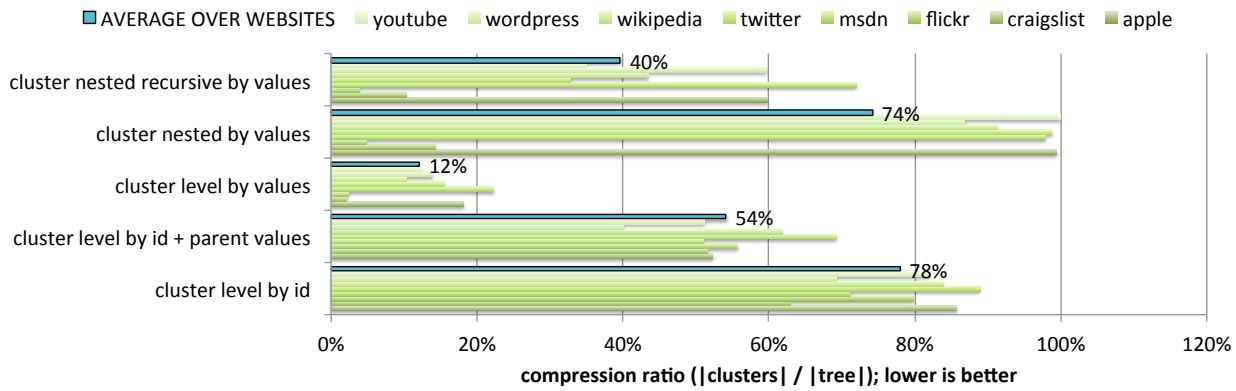


Figure 6.13: **Compression ratio for different CSS clusterings.** Bars depict compression ratio (number of clusters over number of nodes). Recursive clustering is for the reduce pattern, level-only for the map pattern. ID is an identifier set by the C3 browser for nodes sharing the same style parse information while value is by clustering on actual style field values.

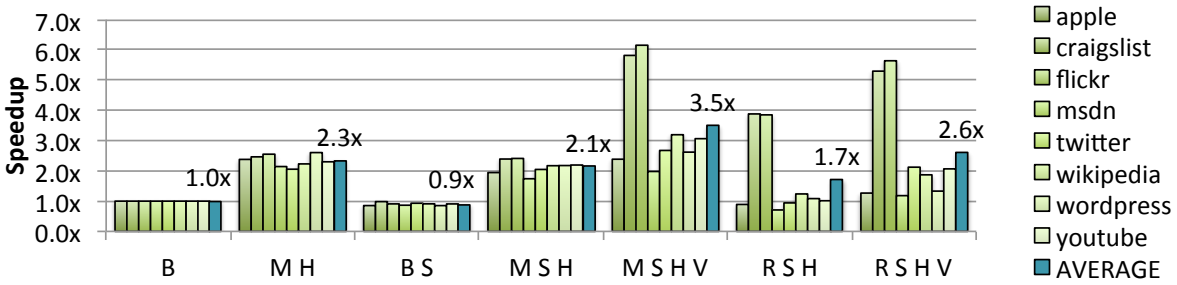
multiple line graphs puts the root node on the first level, the axis for each line graph on the second level, and all of the actual line segments on the third level.

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

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

  Clustering each level by attributes that influence control flow achieved a 12% compression ratio (Figure fig:csscompression): an 8.3X idealized speedup. When we strengthened the clustering condition to enforce stronger invariants in the cluster, such as to consider properties of the parent node, the ratio quickly worsened. Thus, we see that our basic approach is promising for websites on modern subword-SIMD instruction

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



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

Figure 6.14: **Speedups from clustering on webpage layout.** Run on a 2.66GHz Intel Core i7 (GCC 4.5.3 with flags -O3 -combine -msse4.2) and does not preprocessing time.

sets, such as a 4-wide SSE (x86) and NEON (ARM), and the more recent 8-wide AVX (x86). Even longer vector lengths are still beneficial because some clusters were long. However, eliminating all divergences requires addressing control flows influenced by attributes of node neighbors, which leads to poor compression ratios. Thus, we emphasize that 8.3X is an upper bound on the idealized speedup: not all branches in a cluster are addressed.

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

Speedup

We evaluate the speedup benefits of clustering for webpage layout. We take care to distinguish sequential benefits from parallel, and of different clustering approaches. Our implementation was manual: we examine optimizing one pass of the C3 [[CITE]] browser's CSS layout engine that is responsible for computing intrinsic dimensions. The C3 browser was written in C#, so we wrote our optimized traversal in C and pinned the memory for shared access. We use a breadth-first tree representation and schedule for our baseline, but note that doing such a layout already provides a speedup over C3's unoptimized global layout.

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

We first examine sequential performance. Converting an array-of-structures to a structure-of-arrays causes a 10% slowdown (B S in Figure 6.14). However, clustering each level and hoisting computations shared throughout a cluster led to a 2.1X sequential benefit (M S H).

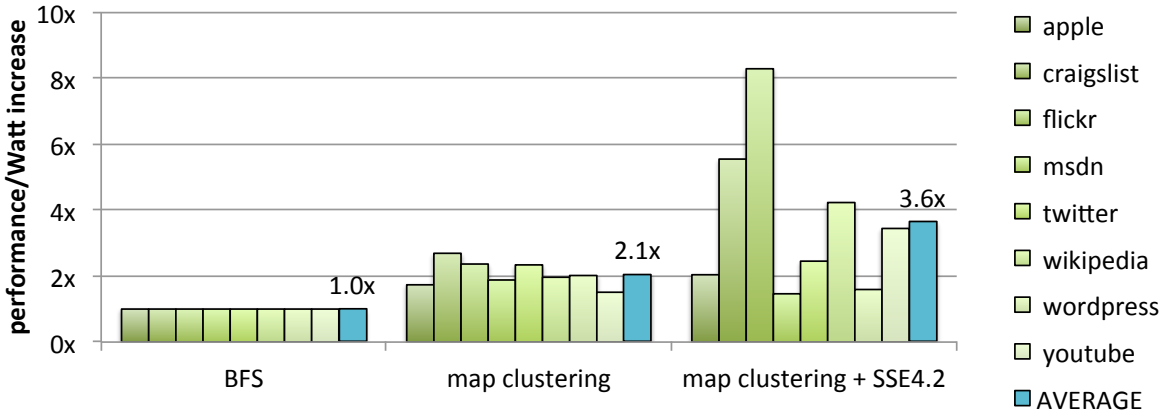


Figure 6.15: **Performance/Watt increase for clustered webpage layout.**

Nested clustering provided more optimization opportunities, but the compression ratio worsened: it only achieved a 1.7X sequential speedup (R S H). Clustering provides a significant sequential speedup.

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

Power

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

In Figure 6.15, we show a 2.1X improvement in power efficiency for clustered sequential evaluation, which matches the 2.1X sequential speedup of Figure 6.14. Likewise, we report a 3.6X cumulative improvement in power efficiency when vectorization is included, which is close to the 3.5X speedup. Thus, both in sequential and parallel contexts, clustering improves performance per Watt. Furthermore, it supports the general reasoning in parallel computing of 'race-to-halt' as a strategy for improving power efficiency.

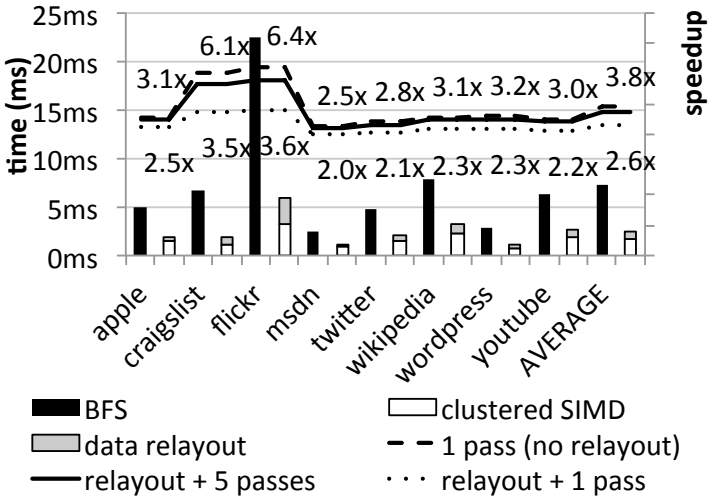


Figure 6.16: **Impact of data relayout time on total CSS speedup.** Bars depict layout pass times. Speedup lines show the impact of including clustering preprocessing time.

Overhead

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

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

For webpage layout, clustering is performed on the client when the webpage is received. We measured performing sequential two-pass clustering. Figure 6.16 shows the overhead relative to one pass using the bars. The highest relative overhead was for the Flickr homepage, where it reaches almost half the time of one pass. However, layout occurs in multiple passes. For a 5-pass layout engine where we model each pass as similar to the one we optimized, the overhead is amortized. The small gap between the solid and dashed lines in Figure 6.16 show there is little difference when we include the preprocessing overhead in the speedup calculation.

6.6 Related Work

1. representation The representation might be further compacted. For example, the last two arrays will have null values for Circle nodes. Even in the case of full utilization, space can be traded for time for even more aggressive compression [[CITE rinard]]

2. sims limit studies

3. duane