Eliminating Tree Irregularities by Clustering

[Not for redistribution]

Leo A. Meyerovich *
University of California, Berkeley
lmeyerov@eecs.berkeley.edu

Todd Mytkowicz
Microsoft Research
toddm@microsoft.com

ABSTRACT

Programmers often increase the performance of their applications by employing data layout optimizations. Unfortunately, control flow can be data dependent, which harms the effectiveness of current techniques. In this paper, we provide novel data layout optimizations for basic patterns of computing over trees based on the idea of *clustering* nodes. We believe clustering is a simple but powerful approach to improving the applicability of hardware and compiler optimizations for many irregular data parallel programs.

Our insight is that we can exploit knowledge about similarities between computations of different nodes to better rearrange traversal order and layout. Nodes are accessed in clusters where each cluster is guaranteed to be self-similar. Focusing on computations over trees, we present clustering strategies for optimizing reductions, parallel iteration with data-dependent control, and search. We primarily focus on how to improve vectorization, but show clustering enables other hardware and software optimizations as well.

We validate our approach by optimizing the *single-core* performance of two challenging and important irregular applications. By characterizing each application according to our patterns and applying the corresponding clustering techniques, we are then able to implement facilitated optimizations such as vectorization. First, we achieve a 4.5x speedup for a phase of the CSS language for laying out webpages. Second, we achieve a 4x speedup for a mature algorithm in computer science, binary search.

1. INTRODUCTION

Computer architects such as Patterson et al. believe hardware performance improvements have hit a brick wall [2] comprised of the *power wall*, *instruction level parallelism wall*, and the *memory wall*. Each wall impacts algorithm performance and design. Supplying power and energy to transistors is increasingly costly: for significant future hardware-driven speedups, algorithms should target parallel architectures. The instruction level parallelism wall shows diminishing returns for techniques such as out-of-order execution: algorithms should target simpler architectures such as multicore and vector. Finally, memory access is increasingly more expensive than computation: algorithms should feature locality and predictable memory access.

Computer scientists and practitioners have long struggled to achieve such architecturally-motivated algorithmic properties. We adopt the "top-down" approach that Patterson et al. have found useful: targeting applications and high-impact *motif* challenge problems distilled from them. We pick two applications (a phase of the CSS language for webpage layout and binary search) and their latent patterns. The patterns are cases of existing motifs:

1. **Parallel iteration with data dependent control** relates to the *graph traversal* motif. We focus on topological tree traversals. Items in a level may be computed independently, enabling MIMD parallelism. However, individual items may diverge based on data-dependent control, challenging finer-grained techniques such as SIMD parallelism, branch prediction, etc.

2. **Reductions** relate to *grid* motifs. A value is computed for each item as a function of neighboring items; the access of values from neighboring items is prominent in the performance profile.

3. **Search** relates to *branch-and-bound* motifs. We examine search down a path of the tree given one input value or vector to compare against each node, such as in binary search and binary decision diagrams, respectively.

Data layout optimizations are a well-studied area that provide techniques for programmers to exploit and increase the regularity of their applications. These optimizations rearrange predictably accessed data into contiguous memory regions. The reorganization of data usually increases spatial or temporal locality. Control flow may also simplify, such as changing a tree traversal from pointer chasing to 1 loop over an array, which enables further hardware and compiler optimizations.

For our case studies, existing data layout transformations such as flattening in NESL [4] for vectorization are ineffective: data dependencies are unaddressed. Consider improving branch prediction for an iteration over a collection where each step executes a data-dependent conditional. Under an arbitrary traversal order, the branch direction is difficult to predict. However, if we partition the collection based on the branch condition, the branch condition for either partition is predictable – it is constant. To improve spatial locality, we might also rearrange the data layout to match the clustered traversal order.

Generalizing, our insight is that the regularities exploited by optimizations such as vectorization and hoisting can be obstructed by data dependencies in the control flow under existing data layout transformations. *Clustering* is a transformation that partitions a data structure according to such

*MSR, ParLab, NSF grants.

dependencies. Given a clustering, we rearrange the traversal order and data layout for a data parallel computation such that evaluation proceeds cluster-by-cluster. The benefit is that, within a cluster, computation is regular and thus optimizable.

In this paper, we contribute the following:

- **The approach of clustering nodes to eliminate data-dependent irregularities.** We present clustering as an effective technique for improving the applicability of a wide range of optimizations for otherwise difficult to optimize programs. For our various patterns, we apply vectorization, hoisting, fission/unswitching, tiling, speculation, and if-conversion, which span both software and hardware optimizations.

- **Clustering optimizations for 3 concrete patterns over trees** for achieving vectorization and other optimizations. We show how to apply clustering to parallel iterations with data-dependent control ("branching") (Section 3), reductions (Section 4), and search (Section 5). We also begin to examine how to *compose* clustering patterns (Section 4). Finally, we analyze the theoretical and achieved performance of cases of our patterns.

- **2 applications: webpage layout and search.** We optimize a phase of the CSS language for webpage layout (Section 6) and binary search (Section 5). By clustering, we can employ vectorization and other techniques. These case studies provide experience reports with irregularities in common applications, how to decompose applications in terms of our patterns, and subtleties in applying clustering. We stress optimizing either application is important in itself.

We first overview of the 3 patterns and existing optimization approaches for them (Section 2) before examining each pattern in depth individually (Sections 3-5). Our case study of binary search is closely related to the search pattern, so we discuss them together. We devote an entire section for vectorizing CSS by first clustering (Section 6).

2. PATTERN OPTIMIZATION OVERVIEW

We briefly overview the 3 tree patterns we target: parallel iteration with data-dependent branching, reductions, and search. For each, we review traditional optimizations to show why they are ineffective and provide our key insight in how to apply clustering to the pattern.

2.1 Iteration with data-dependent branching

Pattern: A commutative iteration over a collection often performs data-dependent branching for handling individual items. The visitor pattern for tree traversals, for example, is a popular approach of representing each traversal as a class containing node handlers that dispatch (branch) based on node type.

Consider the task of vectorizing a MIN/MAX computation over a tree. In a simple sequential implementation (Figure 1), the 2 nested loops traverse a tree bottom-up, level by level. In the loop nest body, depending on the type of a node (type), the value computed (val) is the minimum or maximum of the values of the node's children:

```
#bottom-up tree traversal
for level in tree.intermediateLevels.reverse():
  for i in lvl:
    if (type[i] == MAX):
      val[i] = MAX(val[2i], val[2i + 1])
    else:
      val[i] = MIN(val[2i], val[2i + 1])
```

Figure 1: MIN/MAX bottom-up traversal.

Traditional optimization approach: Consider vectorizing and improving spatial locality. Breadth-first layout achieves spatial locality by placing siblings contiguously, matching the breadth-first traversal order:

$$root() = 1 \qquad left(i) = 2i \qquad right(i) = 2i + 1$$

General segmented scans [5] are vector primitives that can compute the individual minimum or maximum values of multiple arrays at the same time. As input, the elements of 1 array are placed contiguously to form a segment, and multiple segments (arrays) are contiguously allocated to form a segmented vector. To *flatten* trees into segmented vectors, Data Parallel Haskell [19] employs Keller et al.'s encoding [10]: the children of a node form a segment, and a level a segmented vector. This layout is also a breadth-first layout with good spatial locality.

Problem: instruction divergence. We must compute the maximum of some segments and the minimum of others, so we cannot use the same SIMD instructions for all nodes. There are many software, hardware, and hybrid solutions [21, 13] to instead *predicate*. Consider instruction masks, where the vector unit can be programmatically controlled to not compute all inputs to an instruction. A compiler, instead of short-circuiting a conditional, runs both branches, reversing the instruction mask between the branches. Accordingly, in Figure 2, both the maximum and minimum instructions are evaluated for each node but only one of these logical results will be stored. On a 4-wide vector instruction set, 2 segments would be computed over in parallel:

```
for lvl in tree.intermediateLevels.reverse():
  for i in lvl by 2:
    segmented_children = val[2i, ..., 2i + 3]
    #predicated compute
    storeMask([type[i], type[i + 1], 0, 0])
    val[i, 0, i + 1, 0] =
      Max4_segscan(segmented_children)
    storeMask([~type[i], ~type[i + 1], 0, 0])
    val[i, 0, i + 1, 0] =
      Min4_segscan(segmented_children)
```

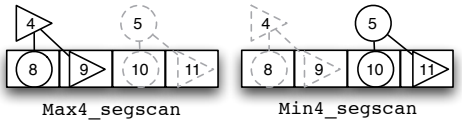


Figure 2: MIN/MAX bottom-up traversal with predication. Faded, dashed nodes are masked. Shape denotes node type.

Predication consumes 2x as many instructions as necessary, masking underutilizes 50% of the available lanes, and

there may be further costs due to gating.

Clustering insight: We can rearrange the nodes in a level based on type. A level is thus a cluster of all MIN nodes then all MAX nodes: evaluation over a cluster requires no predication. Our solution is to cluster globally on dependent data at the data layout step (i.e., during flattening), rather than locally predicate.

2.2 Reductions

Pattern: Parallel iterations over graphs often perform reductions over the neighborhood of each node. Consider the ongoing MIN/MAX example: it computes the minimum or maximum value over the children of each node. The reduction pattern is distinct from the previous branching pattern despite both computations occurring as part of a traversal. Reductions need not branch. Likewise, in the branching pattern, nodes need not communicate with their neighborhoods.

Traditional optimization approach: Segmented scans support scalable reductions, even on trees with low branching factors. Instead of using a vector primitive to reduce the children of 1 node, which only suffices if there are enough child nodes to fill the lanes, multiple segments are reduced in parallel. The amount of parallelism in MIN/MAX, for example, is based on the length of tree levels rather than the branching factor.

Problems with segmented scans: Hardware constraints and data irregularities challenge the traditional approach.

1. **Piecewise parallel primitives.** Blelloch's segmented scan representation places siblings contiguously, but commodity vector architectures do not provide primitives for directly computing over this layout. Instead, hardware primitives are either non-segmented scans or piecewise parallel over two vectors. For general segmented scans such as Max, segmented vectors must be rearranged into a strided layout to use piecewise parallel vector primitives. E.g., all the left children in one array and the right in another.

2. **Varying segment lengths.** Individual reductions of a segmented scan may be of different lengths. Using piecewise parallel primitives to implement them involves predicating the lanes for the shorter reductions: utilization suffers.

3. **Branching.** Consider combining the two patterns of reductions and iteration with branching. Simply clustering to address branching induces an inefficient random memory access pattern on reductions.

Clustering insights: If most accesses to a field are for reductions, we can store it with a *strided* representation for contiguous access by piecewise parallel operators. Furthermore, to avoid masking for simultaneous reductions of different lengths, we can use the number of children of a node as a clustering condition to guarantee equal lengths within a cluster. Finally, we can combine the branching and reduction patterns in a way that achieves efficient memory access by reasoning in terms of composing clustering conditions.

2.3 Tree Search

Pattern: Search is a top-down traversal through a path of a tree. The next node in the path is based on a computation that is performed on the current one.

```
storeMask([1, 0, 1, 0])
for lvl in tree.intermediateLevels.reverse():
 for c in lvl.cluster(type):
  if type[c[0]] == MAX:
   for i in c by 2:
    #gather children
    children = val[lchild[i], rchild[i],
                   lchild[i + 1], rchild[i + 1]]
    #compute
    val[i, 0, i + 1, 0] = Max4_segscan(children)
  else:
   for i in c by 2:
    #gather children
    children = val[lchild[i], rchild[i],
                   lchild[i + 1], rchild[i + 1]]
    #compute
    val[i, 0, i + 1, 0] = Min4_segscan(children)
```

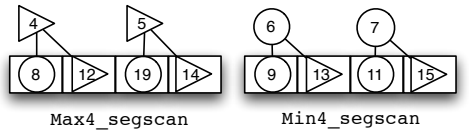


Figure 3: Clustering on branch condition.

Traditional optimization approach: Parallelization is from exploring multiple nodes at the same time, such as, after finding one correct intermediate node, exploring simultaneously searching its subtrees.

Problem: data-dependent traversal Tree search is a difficult pattern for a compiler or hardware to optimize due to the hard to predict traversal pattern over nodes in the tree. Work efficiency is hard to achieve.

Clustering insight: Speculation is an effective technique for dealing with hard to predict data dependencies. We can cluster nodes based on likeliness of succession. For the traversal, speculatively search from any correct node in depth first order if the probabilities are large or non-speculatively in breadth first if not; the choice in traversal strategy dictates the data layout.

3. CLUSTERING PARALLEL ITERATIONS WITH DATA-DEPENDENT CONTROL

In this section, we apply clustering to enabling loop optimizations for parallel iterations with data-dependent control. We provide benchmarks for branch prediction, unswitching, and vectorization to support our claims. Finally, we analyze our clustering strategy: we show a relationship between speedup and the amount of clustering (*compression ratio*) and discuss clustering for standard loop transformations beyond the ones demonstrated.

3.1 Clustering by Branch Condition Value

The optimizations we are interested in are challenged by the branch on MIN/MAX type from one iteration to the next: a data dependency on the iterated item causes instruction divergence across iterations. Our approach is to cluster nodes of a tree level in memory by MIN/MAX type, changing both data layout and traversal order. We rewrite the loop over a level as a loop over different clusters with an inner loop over nodes in a cluster. Important for our optimizations, the previously problematic data dependency

is now constant for the inner loop.

We employ this clustering scheme to enable the optimizations used in Figure 3:

Hardware branch prediction guesses the inner loop branch direction, namely, the node MIN/MAX type. Without clustering, the type is random, so hardware inspection of previous values is ineffective. With clustering, the inner loop is over nodes of the same type, so simply predicting a repeat of the previously seen value is successful on all iterations of the inner loop except the first.

Unswitching lifts the *if-statement* outside of the loop, replacing each branch body with a loop. It is correct because the branch direction is the same for all iterations. A direct benefit is that fewer instructions are retired.

Vectorization can occur without any run time predication due to compile time unswitching: there is no longer a problematic conditional inside the inner loop. Figure 3 still uses masks, but only for storing the result. The vector computation itself runs unmasked.

3.2 Microbenchmarks

Figure 4 benchmarks the performance of the above transformations for MIN/MAX. Our implementations are written in C++ with SSE intrinsics, compiled using GCC 4.5.3 with flags -combine -msse4.2 -O3, and run on a 2.66GHz Intel Core i7 laptop. We use a binary tree with 12 levels to fit the problem in L1 cache. Speedup is relative to a depth-first traversal of a pointer-based implementation. We make data resident in cache by running a traversal once before running a measured trial. We perform 40 trials for each traversal, performing one trial of each algorithm before beginning the next round.

The distribution of the MIN/MAX nodes in a tree impacts the performance of our algorithms so we use several variants. The shade of a bar denotes the distribution. In this section, we consider all MAX nodes (white bars) and a uniform distribution of MIN/MAX nodes (gray bars).

We test hypotheses related to our use of clustering for branch prediction, hoisting/unswitching, and vectorization:

Hypothesis: clustering improves branch prediction. We compare not clustering with level clustering (B S vs. B S L). The primary difference should be in hardware acceleration as instruction and memory access patterns should be similar. We do not expect a speedup for the all-MAX tree because branch prediction should already be effective: indeed, we instead see a slowdown, suggesting a cost to the clustering transformation. For the random tree, speedup increases from 2.4x to 5.7x. The hypothesis is supported.

Hypothesis: clustering enables hoisting/unswitching. While clustering improves branch prediction, hoisting the conditional may further decrease the instruction count. Comparing B S L vs. B S L H for the all-MAX tree, we see speedup increase from 4.5x to 5.6x, and for the random tree, from 5.7x to 7.2x. The hypothesis is supported.

Hypothesis: Predication is insufficient for vectorization. We see a speedup from 5.8x to 11.4x for MIN/-MAX by adding vectorization (B S vs. B S V), seeming to disprove our hypothesis.

The experiment employs a different form of predication than masking: a SIMD comparison of several nodes checks if they are all of the same same type, and if so, uses vector evaluation of the corresponding branch. An upper bound on the speedup is 4x. For an all-MAX tree, the check always succeeds, so, unsurprisingly, there is a speedup from vectorization.

On a random tree, the check fails with probability 92%, preventing vector evaluation. Our hypothesis is supported; vectorization *decreases* the speedup from 2.4x down to to 2.2x.

Hypothesis: clustering enables vectorization. We compare clustering without and with vectorization (B S L H vs. B S L H V). Computation over all-MAX trees slows from a 5.6x to 4.0x total speedup; the random tree slows from 7.2x down to 5.4x. The hypothesis is not supported; clustering by branch condition value is insufficient for the MIN/MAX problem.

Clustering the MIN/MAX computation by branch condition was a bad test of the hypothesis because, as we show in the next section, it can be optimized by considering it as a combination of two different patterns and thereby requires a different clustering approach. Clustering by branch condition does directly enable vectorization in our CSS case study, however.

3.3 Analysis

In this section, we reason about the performance bounds of clustering for parallel iterations with data-dependent control and discuss loop transformations enabled by clustering beyond the ones demonstrates.

3.3.1 Asymptotics

The ability to cluster relates to the speedup achievable by the enabled transformation, implying the quality of clustering is an important optimization target. For predicated vectorization, the *work*, *span*, and speedup of a vectorized traversal relates to the *compression ratio* of clustering:

$$\text{compression ratio} \equiv \frac{|clusters|}{|tree|} = \frac{span}{work} = \frac{1}{speedup} \quad (1)$$

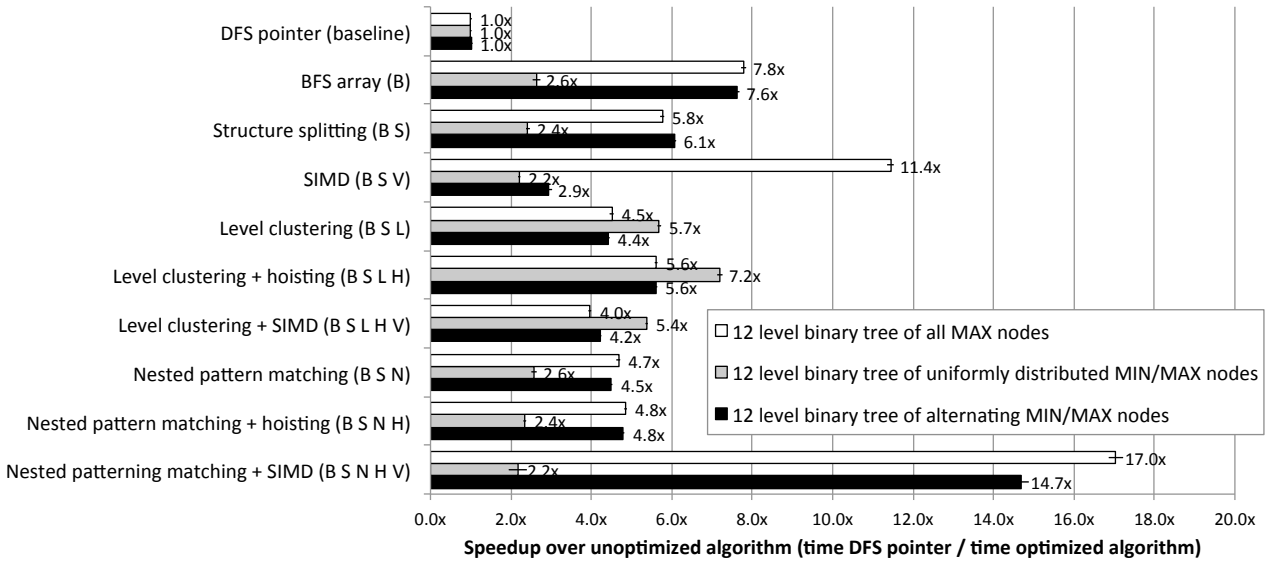
Intuitively, if there is no successful clustering ($|clusters| = |tree|$), the compression ratio is 1 and evaluation is serialized. Likewise, given perfect clustering ($|clusters| = 1$), the ratio limits towards 0 and only a constant number of time steps are needed. The compression ratio and speedup are inversely proportional. Similar reasoning applies to hoisting.

As a sample application, due to the level-by-level evaluation order for MIN/MAX, a lower bound on the compression ratio is $\frac{\log n}{n}$. Such a bound is good because low compression ratios relate to high speedups. In our CSS case study, the challenge becomes achieving the bound: clustering on all branch condition values leads to clusters that are too small, so we must select a subset.

Time spent clustering is an important but separate consideration. E.g., it can often be performed offline at no cost. Alternatively, at run time, each collection can be clustered with linear work and logorithmic parallelization. We found a common misconception is equating clustering with sorting; clustering is simpler. For the example of MIN/MAX, levels of the tree can be clustered independently in parallel, and logorathimically in parallel for each level. Clustering might be performed offline, such as for circuit optimization.

3.3.2 Further Enabled Loop Transformations

Partitioning an iterable collection based on data values enables loop transformations beyond unswitching. In particu-



B = BFS array layout, S = structure splitting, V = vectorization, L = level clustering, N = recursive clustering, H = hoisting

Figure 4: Algorithm speedups over depth-traversal of pointer-based trees under different MIN/MAX assignments. Error bars show the 95% confidence interval for the standard error.

lar, we consider hoisting, software pipelining, tiling, and loop interchange with striding. For example, just as the branch on MIN/MAX type is lifted outside of the loop in unswitching, so can expensive computations be lifted through *hoisting*; we exploit this optimization for our CSS case study.

Loop transformations require static knowledge, such as *unswitching* on branch value, *tiling* on data access patterns, and *software pipelining* on resource usage. If these properties depend on iteration data, traditional implementations of the transformations cannot proceed. Clustering partitions the iteration interval such that the property is known for each cluster, enabling the transformations.

Overall, we see that clustering is a useful tool for improving the applicability of traditional loop transformations, and that the successful application of clustering requires achieving a low *compression ratio*. For the particular case of the MIN/MAX computation, clustering by branch value enables branch prediction and unswitching optimizations, but we require the clustering approach of the next section for effective vectorization.

4. CLUSTERING REDUCTIONS

Our previous clustering solution did not fully optimize the MIN/MAX problem because MIN/MAX is also a *reduction*: a traversal where every per-node computation spends significant time accessing and computing over its small neighborhood. We address 2 challenges for reductions. First, commodity hardware often features piecewise parallel (*vertical*) and subword SIMD operators rather than long arbitrarily segmented scans: we show how to cluster for direct memory access (via striding) and with high utilization (clustering by length). Second, as in MIN/MAX, a reduction may include a conditional, exhibiting the challenging pattern of the last section: we present 2 ways to combine clustering strategies.

In this section, we show how we use clustering to solve these problems (Sections 4.1, Section 4.3.1) and support our claim with benchmarks. Finally, analyzing our approach, we show that as a clustering solution for a pattern involves a *clustering condition*, our 2 solutions of composing pattern optimizations can be cleanly described as operations for composing clustering conditions.

4.1 Strided Vertical Reductions

Hardware may not support the strategy of the last section of placing the children of a span in order and performing a segmented scan over multiple such spans. Instead, segmented scans for commodity hardware can be encoded using *vertical* SIMD primitives, with a corresponding change to a *strided* layout. To lay out the children, first gather all the left children of the spans into one array and all the right children into another. The reduction is then a piecewise-parallel loop over the two vectors (Figure 5). By themselves, these transformations may not be novel, but they are effective and clustering further increases vector lane utilization.

The `Max4_piecewise` instruction utilize its lanes for a piecewise parallel reduction better than the alternative of a non-segmented scan. The piecewise parallel encoding is bounded by the number of segments, which is large in a tree as it is the level length, while the non-segmented scan is bounded by the branching factor, which may be small. Vector store instruction are similarly better utilized. Furthermore, for many instruction sets, the operands to a piecewise-parallel call are two full vectors rather than, for scans, one: the amount of exploitable lanes double by using piecewise parallel instructions.

Generic gathering of the left and right children into their respective vector registers is inefficient because the fetched data might not be contiguous. If most accesses to a field are for reductions, in lieu of a breadth first layout that requires a permutation, we should store the children in a layout that is

```
#bottom-up tree traversal
for_reverse lvl in tree.intermediateLevels:
  for i in lvl by 4:
  #shuffle 2i, ..., 2i + 7 into Left and Right
  L = [val[2i], val[2i + 2],
       val[2i + 4], val[2i + 6]]
  R = [val[2i + 1], val[2i + 3],
       val[2i + 5], val[2i + 7]]
  #compute
  val[i, ..., i + 3] = Max4_piecewise(L, R)
```

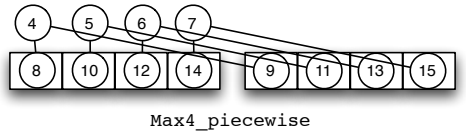


Figure 5: Piecewise-parallel reduction on an all-MAX tree.

already strided or precompute an efficent permutation. The following is one such strided layout:

$$root(i) = 1 \quad left(i) = 2^{\lfloor \log i \rfloor} + i \quad right(i) = 2^{\lfloor \log i \rfloor} + 1 + i$$

4.2 Length-clustered Vertical Reductions

A common irregularity in trees is a non-constant branching factor. Consider a vertical reduction over segments of different lengths. If the reduction performed in a lane is n shorter than the longest reduction, the lane will be masked for n steps. SIMD lane utilization is low.

Clustering based on the number of children eliminates the need for a mask when vertically reducing over the cluster. As all the reductions have the same length, they complete at the same time.

4.3 Dual Reduction/Branching Pattern

We must address both the branching and the reduction patterns in MIN/MAX. Figure 6 extends the clustering optimization of Section 3 for unswitching the conditional with an inefficient application of the vertical SIMD reduction strategy of this section. The problem is that, by only clustering data layout to optimize branching costs, we lose the regular memory access benefit of clustering for reduction access (e.g., striding).

We present two concrete solutions here and generalize them in our analysis:

4.3.1 Recursively conjoining clustering conditions

In our first approach, we want each level to be partitioned by the MIN/MAX type and the children of a homogenous span to be strided. We call these two properties our *clustering conditions*; as they are true at all times, we call them *conjoined*. Such a layout avoids the need for predication and supports vertical reductions with direct memory access.

Recursively clustering is simple and effective dual layout strategy. A level is traversed; any contiguous span of similarly branching nodes such that striding their children yields similar nodes (e.g., all left children are MIN and right children are MAX) is labeled as a cluster and their children are rearranged to be strided. The process then repeats on the next level (with the children that are now strided). By construction, the clustering condition is met. Note that not all

```
for_reverse lvl in tree.intermediateLevels:
  for c in lvl.cluster(type):
    if type[c[0]] == MAX:
      for i in c by 4:
        #gather children
        L = val[lchild[i], lchild[i + 1],
                lchild[i + 2], lchild[i + 3]]
        R = val[rchild[i], rchild[i + 1],
                rchild[i + 2], rchild[i + 3]]
        #compute
        val[i, ..., i + 3] = Max4_piecewise(L, R)
    else:
      for i in c by 4:
        #gather children
        L = val[lchild[i], lchild[i + 1],
                lchild[i + 2], lchild[i + 3]]
        R = val[rchild[i], rchild[i + 1],
                rchild[i + 2], rchild[i + 3]]
        #compute
        val[i, ..., i + 3] = Min4_piecewise(L, R)
```

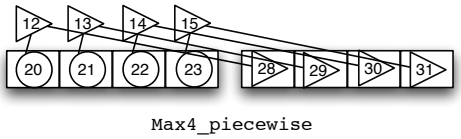


Figure 6: Incorrect composition of branching and reduction clusterings: note random access of children.

nodes are clustered; for every such span of nodes on a level, we restart the process.

Two scenarios motivate applying our recursive clustering scheme to many naturally occurring trees. First, in a nested task parallel computation, tasks of the same type will often spawn tasks of the same type in the same order. Second, in our case study of computing over webpage document trees, document subtrees are often procedurally generated: for example, each item of a shopping cart is encoded as a similarly styled subtree. In both scenarios, our nested scheme applies.

4.3.2 Locally clustering for switchable layouts

For MIN/MAX, a level must be sorted and its children strided, but the level need not be strided nor the children sorted. To exploit this property, instead of requiring a cluster to be simultaneously sorted and strided, we propose to only require that it may be dynamically switched between the two representations. More clusterings are possible so a lower compression ratio may be achieved, and thus higher speedups. Layout depends on access type; the cost of the technique is from demand-driven representation switching. Figure 7 exploits such a representation for MIN/MAX.

A dual clustering is efficiently switchable if the maximum Spearman's footrule distance between the two layouts is under some bound. E.g., if the size of every cluster is under some bound, and each cluster can be locally switched between sorted and strided layouts, then switching is just a local permutation. To make the permutation efficient, part of the generating the clusters should also store the permutation, and permutation should be performed with SIMD operations.

4.4 Experiments

We test our claims about striding, vectorization, and how

```
for_reverse lvl in tree.intermediateLevels:
 for c in lvl.cluster(type):
  clen = c[0].len
  if type[c[0]] == MAX:
   for i in c by 4:
    #gather children
    lmost = lchild[i]
    chldrn = val[lmost, ..., lmost + clen]
    #local permute
    L = permute(children, l_inv_sort[i])
    R = permute(children, r_inv_sort[i])
    #compute
    val[i, ..., i + 3] = Max4_piecewise(L, R)
  else:
   for i in c by 2:
    #gather children
    lmost = lchild[i]
    chldrn = val[lmost, ..., lmost + clen]
    #local permute
    L = permute(children, l_inv_sort[i])
    R = permute(children, r_inv_sort[i])
    #compute
    val[i, ..., i + 3] = Min4_piecewise(L, R)
```

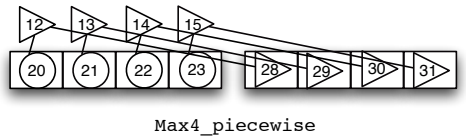


Figure 7: Composing patterns by locally clustering.

to compose clusterings with several experiments.

Hypothesis: striding children optimizes spatial locality. We compare two sequential computations, one without striding (B S) and the other that computes over spans of 4 nodes with strided children (B S N). This is biased towards breadth-first due to smaller working set. For all-MAX trees, speedup decreases from 5.8x down to 4.7x by adding nested clustering, and from 2.4x up to 2.6x from nested pattern matching. The conclusion is indeterminate.

Hypothesis: composing patterns improves branch predication. We rerun the same experiment (B S vs. B S N), also adding a tree of alternating MIN/MAX nodes. Speedup for this new case decreases from 6.1x to 4.5x. The hypothesis is not (immediately) supported.

We suspect this is because the branching pattern for the input, even without clustering, is alternating MIN/MAX nodes: branch prediction hardware should already detect this pattern. Clustering will not improve branch prediction if the branch prediction is already successful. On a random distribution, which is difficult for hardware to support, recursive clustering improves speedup from 2.4x to 2.6x. The hypothesis is supported.

Hypothesis: conjoined clustering enables vectorization. We compare recursive clusterings with hoisting to further adding vectorization (B S N H vs. B S N H V). On a uniformly distributed tree, there is a slowdown from 2.4x to 2.2x: the compression ratio is poor, so we do not expect a speedup. We achieve the highest speedups of all our experiments – up to 17x – when both clustering conditions (perfectly) apply. The hypothesis is supported.

Hypothesis: local clustering enables vectorization. Note shown, we compose the patterns by local clustering, tuning over various allowed distances between the two clustered layouts. A complication is that, even though we constructed the data structure to support switching between clusterings, `x86` does not provide local permutations, and encodings using simpler *shuffle* instructions are inefficient. We instead perform several fetches in a row, hoping that the hardware would detect and optimize the access pattern. We did not observe a speedup – performance approached that of conjoined clustering as the distance increased. The hypothesis is indeterminate.

4.5 Analysis

This section presented bulk vertical reductions with striding, length clustering, and 2 approaches to combining the reduction and branching clustering optimizations. We separately analyze reductions and pattern composition.

Vertical reductions share the theoretical properties of segmented scans: e.g., parallelism is bounded by the tree length, not the branching factor. The encoding is motivated by architectural trends. Clustering eliminates hazards beyond that of traditional striding. E.g., by only striding, reductions of different lengths must be masked. In contrast, clustering on length enables full utilization (up to the number of simultaneous reductions modulo SIMD width). As before, the compression ratio is inversely proportional to speedup.

Our example suggests an approach to composing clustering optimizations. All of our clusterings can be described in terms of a *clustering condition* and the enabled code transformation. Code transformations are often composable (e.g., most loop transformations), but we must still somehow pick a layout. Directly conjoining the conditions may be effective, but, by using demand-driven switching, lower compression ratios and thus higher speedups are theoretically possible. Switchable clusterings may be optimized by both software and hardware: clustering *locally* induces no additional cost in memory access relative to other clusterings (assuming the same compression ratio) and many architectures provide permutation instructions. Reasoning in terms of clustering conditions enables the design of general composition operators.

5. CLUSTERING SEARCH: VECTORIZING BINARY SEARCH

In this section we discuss how to employ the tree search pattern to significantly speed up a mature algorithm in computer science—binary search. A tree search follows a path down a tree, where each visit is determined by the result of a computation on the previous one. We show how to cluster the input array for a binary search based on likely traversal sequences: doing so, we can exploit spatial locality and fine-grained parallelism.

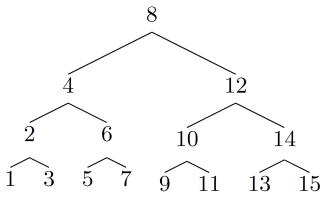
5.1 Binary Search is a Tree Search Pattern

The intuition behind our approach is that binary search visits indices of an input array in a predictable order. Consider the following implementation of binary search:

```
int binary_search(int array[], int n, int key){
  int min = 0, max = n;
  while (min < max) {
    int middle = min + (max - min) / 2;
    if (key > array[middle])
      min = middle + 1;
    else
```

```
        max = middle;
    }
    return min;
}
```

Suppose a programmer calls binary_search with the n=16. On the first iteration of the loop the index variable, middle, is 8; on the second iteration, middle is either 4 or 12. While we may not know the exact value middle will take on the second iteration of the loop, we know it is one of those two values. The third iteration, middle is either $[2, 6, 10, 14]$. To generalize, each level of the following **tree** denotes one of the possible values for middle at each of the first 4 iterations of the binary_search algorithm:



We call such a representation of the indices the *index tree*.

5.2 Clustering for Binary Search

In this section, we show how viewing binary search as a tree search over indices allows us to cluster indices in order to exploit both locality and fine-grained parallelism. The binary search algorithm requires a sorted array as input. We will modify the sorted order using different clusterings.

5.2.1 Breadth first clustering

With breadth first clustering, we organize the sorted array into one that is a preorder traversal over the index tree. For example, the index tree above would be rearranged as:

$$[8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15]$$

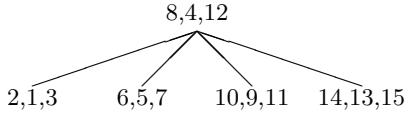
5.2.2 Depth first clustering

With depth first clustering, we organize the sorted array into one that is an inorder traversal over the index tree—given an index i, the left child is at $i+1$ while the right child is at $2^h(i)$ where $h(i)$ gives the depth of the tree rooted at i. The index tree noted above would be rearranged as:

$$[8, 4, 2, 1, 3, 6, 5, 7, 12, 10, 9, 11, 14, 13, 15]$$

5.2.3 Level clustering

With level clustering, we first group N levels of the index tree, which produces an index tree with k elements per node and $k+1$ children per node. This representation is akin to a B-tree. For example, with $k = 3$ our clustering of the index tree noted above is:



This representation is then flattened to an array as with breadth and depth first clustering. For example, the index tree noted above would be represented by the following array:

$$[8, 4, 12, 2, 1, 3, 6, 5, 7, 10, 9, 11, 14, 13, 15]$$

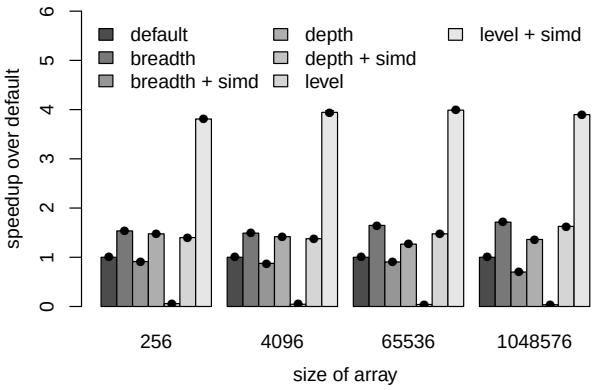


Figure 8: Speedup for clustered binary search.

These clustering techniques are designed to exploit regularity in the traversal patterns as the binary search algorithm walks level by level through the index tree. We exploit cache locality, and, as we show in the next section, they allow us to search in parallel using vectorization.

5.3 Vectorization through predication

Broadly, we vectorize by speculatively perform multiple comparisons in parallel. We use a different predication strategy for each of the various clusterings:

5.3.1 Breadth and Depth first predication

Given an input array

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]$$

we split the array into four sub-arrays: $[1, 2, 3, 4]$, $[5, 6, 7, 8]$, $[9, 10, 11, 12]$, and, $[13, 14, 15]$. Then, for any key, we search each sub-array in parallel using SIMD vector units. Each subarray has a 25% chance of containing any input key so our theoretical speedup is 4X faster than the sequential binary search algorithm.

Breadth first predication works on the breadth first clustering, while depth first predication on the depth first clustering.

5.3.2 Level predication

Level clustering places N levels of an array into contiguous locations in memory. With level predication, we execute N levels of the tree with a single SIMD operation. Because the k data elements that comprise the N levels are contiguous, we only need a single vector load to the data. For example, given the level clustering noted above, we can compare a key against the the values at $[8, 4, 12]$ with one SIMD comparison. We will discard one of those comparisons (e.g., if $key = 3$, whether $k < 12$ does not matter as we will move to the child $[2, 1, 3]$) in the index tree.

5.4 Evaluation

We describe the platform we used to run our experiments, the methodology we used to measure our system, and the details of how we implemented each of the high level clustering techniques described above.

5.4.1 Methods

We conducted our experiments on an Intel Core i5 workstation (2.66GHz processor with 8G of RAM and 4 wide SIMD units).

We optimized each version of binary search individually; our default implementation is hand-optimized code and thus is already a fast version of the algorithm.

We used the Microsoft Visual Studio C++ compiler (2010) with full optimizations (/Ox). We evaluate over 4 different input array sizes (2^8,2^{12},2^{16}, and 2^{20}). The array is filled with random integers and then searched with 100K random keys. We report the number of machine cycles for 30 runs.

5.4.2 Implementation of Binary Search Algorithms

We use breadth first and depth first clustering as noted above. Our implementation of level clustering groups $N = 4$ levels at a time (so $k = 16$).

For depth first clustering, we do not calculate the right child at run time as it requires we do a costly floating point power operation. Instead, we have an array that, for index i, provides the index of the *right* child (the left child is always $i + 1$ in a depth first layout.

5.4.3 Better locality with Clustering

Clustering produces faster algorithms because of better cache locality. Figure 8 plots the speedup (running time / optimized running time) along the y-axis for our three clustering techniques (x-axis) for different sized input arrays. Each bar in the bar plot gives the mean (whiskers denote 95% confidence interval of the mean).

To summarize our results, for the largest array size, breadth first clustering provides a speedup of 1.71x, depth first clustering 1.36x, and, level clustering 1.62x.

The reason breadth first clustering provides the best speedup is worth noting; the indexing scheme for a breadth first layout is simple (given an index i, the left child is at $2i$ while the right child is at $2i + 1$). Despite that a breadth first layout does not provide good data locality (e.g. when i gets large $2i$ is unlikely to be physically located near i), the hardware was likely able to prefetch the left and right children for any given index i because it is an arithmetic operation.

5.4.4 Vectorization with Clustering

Vectorization *sometimes* produces faster algorithms. For the largest array, breadth first clustering + SIMD provides a *slowdown* of 0.69x, depth first clustering + SIMD provides *slowdown* of 0.03x, and level clustering + SIMD provides *speedup* of 3.9x.

The reason that breadth first + SIMD and depth first + SIMD are so slow, despite that they are searching only a quarter of the array, is because memory loads serialize the SIMD unit, thus limiting parallelism. On the other hand, level clustering organizes its data in contiguous regions of memory, thereby removing the impact of memory serializing the SIMD unit. This is the reason level clustering + SIMD is almost 4X faster than the default implementation.

5.5 Summary

In this section, we demonstrated how to apply three clustering techniques—breadth first, depth first, and level—to a mature algorithm in computer science, binary search. Aided by clustering, we show a significant (3.98X) single-core speedup.

6. EXPERIMENTS

We performed a case study of manually optimizing part of the CSS layout engine in the experimental C3 web browser [12]. Fast and power efficient web browsers are an important topic due to the ubiquity of web browsers and mobile computing. Perhaps not obvious due to popular interest in optimizing JavaScript interpreters [9], JavaScript only accounts for 10-20% of the time [16] for common browser workloads. Native libraries must also be optimized, such as parsing, rendering, and examined in this work, layout solving.

In this section, we detail our experiences with characterizing CSS layout as a combination of the iteration with data-dependent branching pattern and the reduction pattern. We achieved a 4.5x speedup mostly from hoisting, vectorization, and breadth first layout. We first describe the CSS layout computation and our experimental setup before continuing to our experiences with optimizations enabled by clustering and clustering heuristics.

6.1 CSS Layout Computation

A document layout engine typically performs a sequence of passes over a tree. A traversal visits each node, dispatching to distinct handlers based on the traversal and the node type, similar to the *visitor pattern*. As found by Meyerovich et al. [16], the traversal for CSS is data parallel; it matches both our data-dependent branching (Section 3) and reduction (Section 4) patterns. We examine one representative pass that is a bottom-up traversal. It computes the preferred minimum and maximum widths of each node by examining the corresponding values of the node's children and various node-local style constraints.

6.2 Experimental Setup

C3 is written in C#, which does not support vectorization, so we translated the layout code into C++. To lower environmental noise, we recorded the input sent to the solver in a file in order to run our C++ code in isolation. To simplify implementation, we further deviate by storing node data as fields (or structure split fields) rather than hash tables.

Our measurements are over 5 popular websites (YouTube, Wordpress, Wikipedia, Twitter, MSDN, Flickr, Craigslist, Apple) whose dynamic content has been manually removed. For each site, we perform 40 trials, where one trial consists of looping through all of the optimized layout strategies. Before each measurement, we attempt to make the tree resident by running the computation once.

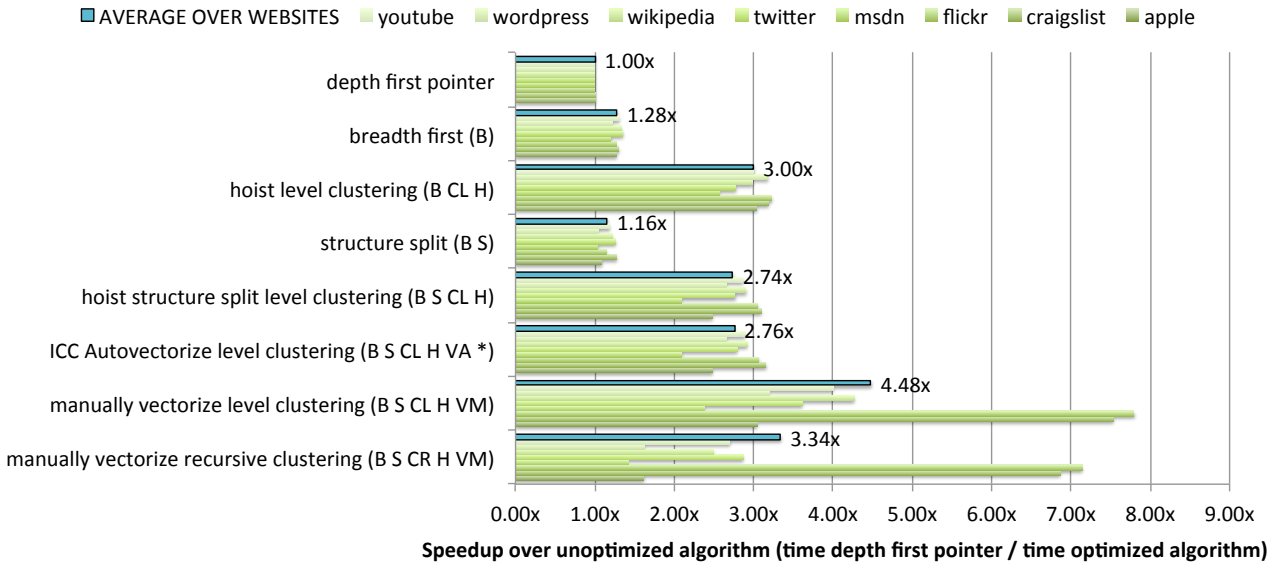
All measurements are on a 2.66GHz Intel Core i7 laptop. We use GCC 4.5.3 with flags -O3 -combine -msse4.2, except for computations employing automatic loop vectorization, where we use ICC 12.0.4 with flags -O3 -ipo -fno-alias -xSSE4.2 -restrict and the pragmas ivdep and always.

Note that measurements are of traversal time, which does not include clustering time. We discuss why in our summary.

6.3 Hoisting and Spatial Locality

Overall, we see significant speedups from hoisting and breadth first layouts (these and others are depicted in Figure 6).

The benefit of spatial locality is not as high as in our microbenchmarks. Adding a breadth first layout yields an average 1.28x speedup (depth first pointer vs. B). Adding structuring splitting (B vs. B S) lowers the speedup to 1.16x. As we only serialize and store the fields relevant to our computation, structure splitting might actually yield a speedup



B = breadth first array, S = structure splitting, H = hoisting, CR = cluster recursively, CL = cluster by level, VA*= automatically vectorize loops with the Intel compiler, VM = manually vectorize loops with compiler intrinsics

Figure 9: Speedups of various clustering strategies and enabled optimizations for one pass of laying out popular websites.

when multiple passes are supported: the current computation happens to use most field instances.

Hoisting over a clustered tree more than doubles performance. Without structure splitting, hoisting improves speedup from 1.28x to 3.00x (B vs. B CL H). With structure splitting, the speedup improves from 1.16x to 2.74x (B S vs. B S CL H).

Overall, even without vectorization, clustering significantly improves the efficacy of traditional optimizations.

6.4 Vectorization

Vectorization yields our best cumulative result: a 4.5x speedup. It achieves 1.6x *relative* speedup over not vectorizing (B S CL H vs. B S CL H VM). We do not expect a 4x relative speedup because many computations are already successfully hoisted out of the inner loop. To determine the quality of inner loop vectorization, we measured just these loops: manual use of intrinsics often did much better than autovectorization.

6.5 Clustering Heuristics

Designing a clustering heuristic is an important optimization step because a low compression ratio yields a high speedup. For example, for our fastest algorithm (B S CL H VM), the compression ratio is strongly negatively correlated with the speedup at -0.96. Figure 6 shows the compression ratios of different schemes over various websites.

Two basic considerations featured in our selection of a clustering heuristic: which clustering scheme to apply and how to instantiate it.

Choosing a clustering scheme. It is important to target the right pattern. For example, we presented the recursive, *conjoined* clustering for a combined pattern: the compression ratio is only 40%. To avoid leaving much of the tree unclustered, we use a hybrid, applying level clustering for the remaining nodes (and invoke the corresponding SIMD handlers). The clustering ratio is still worse than from not considering the reduction pattern; speedup suffered. Furthermore, most of the CSS computations are node-local. We believe both reasons led to poor speedups when incorporating the reduction pattern.

Instantiating a clustering scheme. How to match two nodes as belonging to the same cluster is more complicated than we originally envisioned. The layout engine already provides a node type hierarchy (e.g., different types for word-wrapping vs. images), but many nodes with similar behavior are described by the same type. To handle the differences, the shared handler repeatedly inspect different node attributes and then branches. Clustering based on the existing types thus fails to address many branch conditions; we did not implement this clustering heuristic.

We implemented several clustering heuristics, including:

1. **IDs.** Fields are copied from set objects, so we clustered nodes by the ID of their originating sets. This scheme yields a poor 78% compression ratio.

2. **IDs with parent fields.** A node often branches based on a value of a parent node; the preceding ID-based clustering was actually from examining both the parent ID and node ID. As few parent fields impact control flow, in lieu of the parent ID, we select several parent fields. The compression ratio dropped to 54%.

3. **Fields.** Simple yet effective, we compare all the field values of nodes. The compression ratio is a low 12%. Note that some sites compress to 1-2% while others are closer to 20%.

6.6 Summary

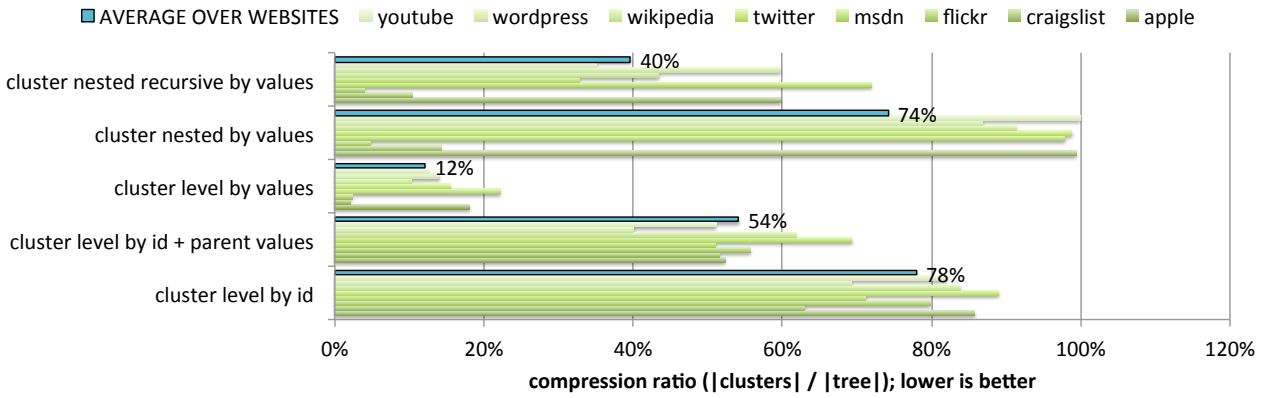


Figure 10: Compression ratios of different algorithms for popular websites.

Overall, we have shown that clustering enables significant optimizations such as hoisting and vectorization for a computation as irregular as CSS. Picking the appropriate clustering heuristic is an important but difficult part of this process. Downstream optimizations may still be difficult; e.g., clustering facilitates loop vectorization, but we still found ourselves manually vectorizing loops.

Our measurements do not include clustering time, which is important as this particular application requires run time clustering. Clustering seems optimizable in theory; it is linear, embarrassingly parallel, and likely vectorizable. We chose not to implement it as a fair characterization would be to build a parallel CSS and HTML parser for C3 in which to integrate clustering; this seemed excessive for determining whether we can vectorize a tree computation. .

In summary, clustering is effective: our average *single-core* speedup is 4.5x, and, in two cases, 7.5x.

7. RELATED WORK

We cluster computations over trees where control flow often depends on data and emphasize efficient vectorization. Most related work focuses exclusively on data layout optimizations for spatial and temporal locality or for vectorizing simple control flow patterns over data structures.

7.1 Data layout optimizations

Chilimbi et al. organize a tree in memory to better exploit the cache hierarchy's [6]. They cluster data by access pattern: resources are *coallocated*, meaning they are placed in memory to improve spatial locality. We cluster for spatial locality, but also change the access order itself. Furthermore, we target vectorization, which impacts our clustering approach, layout, and even the ensuing compression ratio.

Frigo et al. optimize cache-oblivious algorithms [8] so that algorithms are not effected by hardware parameters such as cache line size. We must further optimize for vectorization and find tuning to be an effective technique.

Nuzman et al. automate data layout optimizations for vectorization such as striding [18]. Many such optimizations are known, e.g., array-of-structures to structure-of-arrays; our challenge is finding the equivalents for our patterns.

7.2 Tree Optimizations

Matsuzaki et al. study parallelism in tree reductions [15] but, despite presenting an implementation, do not report any speedups. Matsuzaki et al. also discuss a set of tree patterns that operate on rose trees [14]; data parallelism is effective for large trees over multiple processors. We also target abstract operations over trees (e.g., reductions and search), but exhibit similar speedups *while still using the same processor*.

Vectorization is particularly difficult. Indicative of the challenge, Barnes provides an early algorithm in a paper titled "A modified tree code: Don't laugh; It runs" [3]. He targets a different pattern of handling multiple queries, such as for n-body simulations. Recently, Kim et al. develop FAST: a parallel version of an in memory tree structured index [11] for both the CPU and GPU. We both target the tree search pattern, except while FAST searches 2 levels of a tree at a time, we search 4. Chatterjee and Blelloch develop a theory of segmented scans [5]; we compare to it in Section 4.

Our application of CSS is inspired by Meyerovich et al. [16], who show that layout is data parallel over a tree, but only explore MIMD optimization. We believe these approaches can be combined. Likewise, we suspect Zhang et al.'s memoization of CSS layout [23] can be improved by clustering.

7.3 Language support

Many languages and frameworks automate vectorization and other optimizations on data parallel computations. While automation is beyond the scope of this work, these systems feature relevant support for irregular computations.

Reps proposes to vectorize tree reductions in *scan grammars* [20]. Blelloch's NESL [4] achieves vectorization of structures with a bounded depth. Keller et al. generalize NESL to recursive types (e.g., trees of arbitrary depth) [10], which Data Parallel Haskell [19] employs. Unfortunately, the irregularity challenging scan grammars – nodes of different types – has not been addressed until our work.

Likewise, we found benefits in changing the implementation of segmented scans, which are the assumed vector primitives in these systems. We suspect the general code transformations of these systems are still applicable to many of our optimized layouts. Our approach of striding children might be *manually* used under Keller's type-directed flatten-

ing [19] by encoding children values as fields of the parent node; analysis could then further automate it.

Data dependencies are visible to just-in-time compilers. For example, tracing can specialize on common values [9]. Data structure support is difficult; of note are recent advances in incrementalization [7, 1]. Intel Array Building Blocks [17] JIT vectorizes structures such as graphs, but we are not aware of support for the patterns we examined.

Recently, Zhang et al. [22] show that permuting arrays improves memory access patterns for irregular GPU programs. Our work is in a similar spirit, though we focus on trees rather than arrays, data-dependent control, and basic optimizations such as hoisting.

8. CONCLUSION

In this paper, we have examined how to optimize several irregular computations on trees which are dominated by data dependent control flow. Our insight is that clustering data enables traditional hardware and software optimizations that are currently inapplicable—most notably, vectorization. We decompose computations as patterns over trees and show how to organize data to enable hardware and software optimizations.

We validate our approach by discussing how to apply clustering on two case studies: binary search and a phase of the CSS language for webpage layout. Binary search is a well-studied and commonly used algorithm in computer science—we demonstrate a single-core speedup of 4x. Due to low-powered mobile devices and responsiveness requirements, web browsers are sensitive to performance—we demonstrate a single-core speedup of 4.5x for a phase of CSS.

9. ACKNOWLEDGEMENTS

In no particular order, Dan Grossman, Herman Venter, Wolfram Schulte, Rastislav Bodik, Nikoli Tillman, Christopher Batten, Jim Larus, and Bryan Catanzaro have provided useful feedback throughout this project.

10. REFERENCES

[1] U. A. Acar, G. Blelloch, R. Ley-Wild, K. Tangwongsan, and D. Turkoglu. Traceable data types for self-adjusting computation. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 483–496, New York, NY, USA, 2010. ACM.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] J. E. Barnes. A modified tree code: don't laugh; it runs. *J. Comput. Phys.*, 87:161–170, March 1990.

[4] 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.

[5] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Proceedings Supercomputing '90*, pages 666–675, Nov. 1990.

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

[7] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *In European Symposium on Programming*, pages 294–308, 2006.

[8] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–, Washington, DC, USA, 1999. IEEE Computer Society.

[9] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM.

[10] G. Keller and M. M. T. Chakravarty. Flattening trees. In *Euro-Par*, pages 709–719, 1998.

[11] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 339–350, New York, NY, USA, 2010. ACM.

[12] B. S. Lerner, B. Burg, H. Venter, and W. Schulte. C3: an experimental, extensible, reconfigurable platform for html-based applications. In *Proceedings of the 2nd USENIX conference on Web application development*, WebApps'11, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association.

[13] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28:39–55, March 2008.

[14] K. Matsuzaki, Z. Hu, and M. Takeichi. Parallel skeletons for manipulating general trees. *Parallel Comput.*, 32:590–603, September 2006.

[15] K. Matsuzaki, Z. Hu, and M. Takeichi. Towards automatic parallelization of tree reductions in dynamic programming. In *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '06, pages 39–48, New York, NY, USA, 2006. ACM.

[16] 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.

[17] C. J. Newburn, B. So, Z. Liu, M. D. McCool, A. M. Ghuloum, S. D. Toit, Z.-G. Wang, Z. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *CGO*, pages 224–235. IEEE, 2011.

[18] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. In *Proceedings of the*

2006 ACM SIGPLAN conference on Programming language design and implementation, PLDI '06, pages 132–143, New York, NY, USA, 2006. ACM.

[19] S. Peyton Jones. Harnessing the multicores: Nested data parallelism in haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 138–138, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] T. Reps. Scan grammars: parallel attribute evaluation via data-parallelism. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, SPAA '93, pages 367–376, New York, NY, USA, 1993. ACM.

[21] J. E. Smith, G. Faanes, and R. Sugumar. Vector instruction set support for conditional operations. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 260–269, New York, NY, USA, 2000. ACM.

[22] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. *SIGPLAN Not.*, 46:369–380, March 2011.

[23] K. Zhang, L. Wang, A. Pan, and B. B. Zhu. Smart caching for web browsers. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 491–500, New York, NY, USA, 2010. ACM.