Clustered Data Parallelism

Leo A. Meyerovich
UC Berkeley
lmeyerov@eecs.berkeley.edu

Todd Mytkowicz
Microsoft Research, Redmond
toddm@microsoft.com

Abstract

Many data layout optimizations cluster data accesses and memory into high-locality groups in order to optimize for the memory hierarchy. In this paper, we demonstrate that similar clustering program transformations enable efficient vectorization. We call this approach clustered data parallelism (CDP). CDP enables fast and power-efficient parallelism by partitioning a data structure into clusters such that SIMD evaluation is efficient *within* a cluster.

We describe the CDP latent in three common computational patterns: map, reduce, and graph traversals. Demonstrating the benefits of CDP, we present case studies of instantiating the CDP patterns in order to design fast and power-efficient binary search and webpage layout algorithms. First, we increase binary search SIMD scalability by using CDP to expose speculative parallelism. Second, we achieve the first SIMD webpage layout algorithm by using CDP to eliminate heavy branching.

We report strong performance improvements. Targeting AVX, we see a 5.5X speedup and 6.9X performance/Watt increase over FAST, the previously fastest SIMD binary search algorithm. Running webpage layout with SSE4.2 instructions, we observe a 3.5X speedup and 3.6X performance/Watt increase over an already optimized baseline.

1. Introduction

Many data layout optimizations cluster data accesses and memory in order to optimize for memory hierarchies. In this paper, we instead apply clustering to optimize for a different hardware resource: SIMD units.

Parallelism poses an opportunity for improving the performance per Watt of phones, laptops, and servers [2]. Same instruction multiple data (SIMD) architectures are particularly attractive because they amortize instruction and memory costs across many data operations. Unfortunately, while SIMD is more efficient than multiple instruction multiple data (MIMD) evaluation, it is also more constrained [17]. As a consequence, SIMD units often remain unused.

We present new SIMD patterns by adapting *clustering* data locality optimizations. Locality optimizations largely descend from external sort algorithms [25] that optimize for the high cost of memory access. The variants we are inspired by optimize cache accesses by transforming loops to access data in high-locality groups. As examples, tiling [13] and co-allocation [10, 11] exploit the commutativity of computations and data allocations to rearrange the access order and layout of data into blocks. Computations proceed block-by-block, where evaluating a block of nodes primarily accesses data in that block. Jo and Kulkarni [14] apply similar reasoning to caches in multicore hardware.

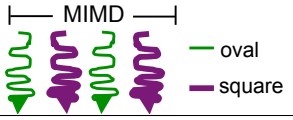
Data layout transformations are already used for vectorization, but with little emphasis on clustering. For example, nested data parallel (NDP) language such as NESL [6, 26] can convert an array of pairs into a pair of arrays that are more amenable to SIMD access. However, NDP does not lead to SIMD speedups when the instructions across different data elements diverge, the data elements are not accessed in parallel, or they are not accessed in order.

Our insight is that, instead of vectorizing an entire loop, we can find SIMD-friendly subintervals. Locality clusterings already guarantee ordered access on arrays and our clustering patterns further guarantee that tasks in a cluster use the same instructions and can run in parallel. These three properties improve vectorization.

We call such evaluation *clustered data parallelism* (CDP). It is not fully data parallel because cross-cluster evaluation may still be performed sequentially, such as due to instruction divergence. To design CDP algorithms, we reason about the size of clusters and the efficacy of SIMD evaluation within a cluster.

We present computational patterns that benefit from CDP and apply them to two case studies: manually vectorizing webpage layout and binary search. These case studies use three of the 13 patterns found as necessary by Asanovic et al. [2] for general purpose parallel systems. Webpage layout is a node-labeled reduction (*map and reduce patterns*) which is further complicated by an irregular tree structure and heavy predication; clustering load balances work and eliminates instruction divergence. Binary search matches the *graph traversal pattern*, with a loop-carried dependency on node order that we break by speculatively clustering.

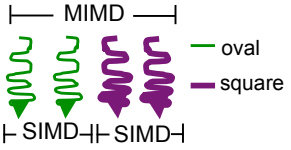
As an example, the pattern we use to vectorize webpage layout also applies to parallel method dispatches:




```
@parallel [shape.rotate() for shape in shapes]
```

The problem is that different Shape subtypes will cause rotate() calls to dispatch to different methods. As indicated by the parallel instruction trace, this is fine for MIMD evaluation. However, traditional vectorization [28] would run all of the possible methods for each datum, and upon completion, only store the result of the correct ones. This is slow and performs useless work.

Our solution is simple: cluster by Shape subtype.




```
@parallel for cluster in shapes.split(typeof):
  if typeof(cluster[0]) == Oval:
    @vector [Oval.rotate(shape) for shape in cluster]
  elif typeof(cluster[0]) == Square:
    @vector [Square.rotate(shape) for shape in cluster]
```

The clustering guarantees monomorphic dispatch within the inner loop, saving time and avoiding useless work. Our patterns reuse this idea of partitioning into SIMD-friendly clusters. Through clustering, they also improve load balancing and support out-of-order traversals.

In summary, this paper introduces clustered data parallelism. In CDP, data accesses and memory are clustered in order to optimize for SIMD evaluation within a cluster. To aid developers in designing CDP algorithms, we present map, reduce, and graph traversal CDP patterns inspired by clustering transformations originally designed for increasing locality.

We manually applied our patterns to vectorizing webpage layout and binary search. First, we present a webpage layout algorithm that exploits our map pattern. Second, we present a binary search that exploits our graph traversal pattern. We report a 3.5X speedup and 3.6X performance/Watt increase over an already optimized webpage layout baseline. We achieve a 5.5X speedup and a 6.9X performance/Watt increase over the previously fastest binary search algorithm, FAST [16].

2. Three CDP Patterns

We present our map, reduce, and out-of-order traversal algorithms as parallel patterns and use the pattern presentation structure pioneered by Mattson et al. [21]. For each pattern, we discuss the parallelization problem being addressed, the context in which to use the pattern, forces impacting use, our core solution, and common variations. We analyze the patterns in later sections.

2.1 Out-of-order traversals: hot path clustering

A common pattern is traversing a collection and computing a value for each item. SIMD evaluation over multiple items is difficult if the computation for one item determines the next item to traverse. We can apply clustering when the probability of which item to traverse given the previous one is sufficiently strong.

Context. Out-of-order traversals often appear in data structures. For example, binary search traverses a tree to find a node with some desired value. A simple loop suffices for a tree with a breadth-first layout:

```
1 i = 0
2 while key[i] != v:
3   i = 2 * i + (1 if key[i] < v else 2)
```

Each iteration performs a comparison in order to find the node to examine for the next iteration.

Parallel evaluation of individual loop iterations is challenged by a loop-carried dependency on i, the node to examine. However, our vectorization pattern can be applied if there is knowledge of hot subpaths through the collection. We will examine in detail the simple case of a left bias in the comparisons.

Impacting forces. The scalability of our core pattern is a function of the predictability of hot subpaths.

Extending the pattern to more complex operations over richer structures may introduce additional costs. For example, if the comparison operator is extended to inspect the values of child nodes, our reduce pattern (Section 2.3) can be combined with this one. The composition is subject to limitations of both approaches.

```
1  i = 0;
2  for bnodesBelow in [9, 1, 0]:
3    @vector m = [ val < k[i + o] for o in range(0, 4) ]
4    @vector check = (m == [ True, True, True, True ])
5    if check:
6      i = i + 16                          /* next hot subpath */
7    else:
8      iStart = i
9      track = 0
10     for j in range (4, 0, −1):
11       track = track + ( (1 << j) if val > k[i] else 0)
12       i = i + (1 if val < tree.data[i] else (1 << (j − 1)) )
13     i = iStart + (1 + track ∗ bnodesBelow) ∗ 16
14 /∗ ... handle the last iteration sequentially ... ∗/
```

Figure 1: CDP evaluation of binary search. The vector length is 4, tree depth is 13, and comparisons bias left.

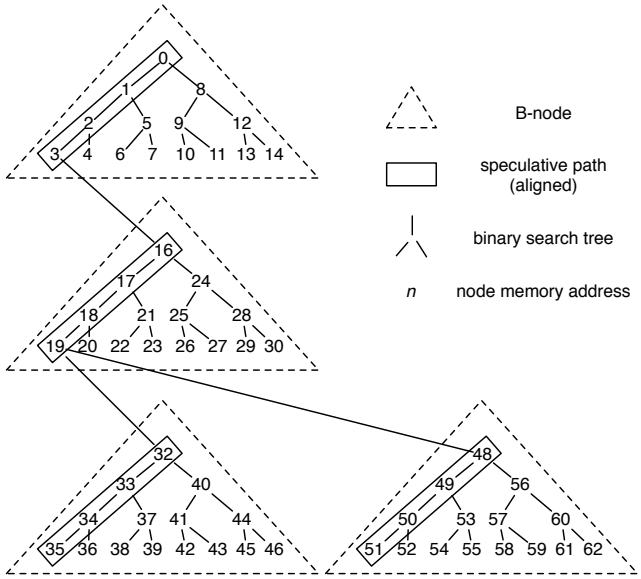


Figure 2: Clustered tree layout for binary search. The tree depth is 12 and data blocking is for an architecture with vector and cache line length of 4.

Solution structure. Our insight is that we can adapt a locality clustering that prefetches hot subpaths to now also compute in SIMD over the hot subpaths. Each loop iteration runs the same instruction, so SIMD instructions can compute across the cluster. For the typical case, a SIMD check that the expected result was achieved suffices for proceeding to the next cluster.

Without loss of generality, consider a binary search where comparisons bias to the left. The first 8 comparisons would most likely be for breadth-first nodes

[0, 1, 3, 7] followed by [15, 31, 63, 127]. Shown in Figure 2, we cluster the tree as a B-tree [5] where each B-node represents a subtree of depth four in order to match a vector length of four. Crucially, we use a preorder layout for both the allocation of different B-nodes and the layout of nodes within a B-node.

Our clustering can be first understood as a speculative data locality optimization that arranges data according to hot subpaths, similar to that of Chilimbi et al. [10] and Ding and Kennedy [11]. The first four likely comparisons, corresponding to the logical nodes above, are for locations [0, 1, 2, 3]. The next most likely four are for the first locations in the next B-node in memory: [16, 17, 18, 19]. Data loads benefit from the speculative layout because, for example, when hardware fetches location 0, it also prefetches locations 1, 2 and 3. Likewise, the fetch of location 16 also speculatively prefetches 17, 18, and 19. Misspeculating the traversal order will load data from different cache lines, but the cache lines are still from the same B-node.

We adapt the clustering for efficient SIMD evaluation. Instead of just speculatively prefetching data for upcoming iterations, our pattern is to speculatively compute over them in parallel as well. In the case of a left bias, our algorithm in Figure 1 first performs all of the comparisons for locations [0, 1, 2, 3] using just 1 SIMD instruction (line 3), and then checks that the result is [True, True, True, True] (all less than) with another (line 4). On success, the search proceeds to speculatively compute over the next B-node in memory, etc. On failure, a sequential search is used to find the correct path through the B-node, after which speculative execution resumes for the next B-node. For example, if the second SIMD comparison failed with result [True, True, False, False], recovery would sequentially traverse locations [16, 17, 18, 20] and then either jump to the third or fourth child B-node.

Several optimizations are useful in practice. First, we pad B-nodes so that SIMD loads are aligned. Second, we store cold paths alongside hot paths in B-nodes in order to lower misspeculation costs. Third, we do not represent the last levels of the tree as B-nodes, decreasing memory consumption. These are analogous to optimizations for the sequential case. Kim et al. [16] describe complementary out-of-core optimizations.

Generalization. Our pattern generalizes beyond left bias predictions. The preorder descendent of a node (and B-node) is simply the most likely child to be traversed next. The hot paths therefore follow the same memory layout as in Figure 2. Consequently, instead of expecting a SIMD comparison of [True,True,True,True], line 4 of Figure 1 generalizes to a check against the bit string of the expected path.

Algorithms beyond binary search can be encoded with our pattern. For example, beyond the scope of this paper, we also optimized the decision trees used by a commercial "web-scale" AdaBoost recommendation system. Instead of comparing each node against the same value val, there is a different key to compare against each individual node. As another variant, our pattern may be applied to graph traversals by picking a likely minimum spanning tree or using the common encoding of an infinite tree by duplicating nodes.

2.2 Maps: branch condition clustering

Similar tasks are often independently performed on all of the items of a collection. Vectorizing parallel tasks that run evenly slightly different instructions is challenging. Our intuition for CDP is that clustering data and code based on predictable branch conditions prevents divergence within a loop over a cluster.

Context. A common occurence of the map pattern is in list comprehensions. List compressions are syntactic forms, such as used by the following Python code:

```
[ i + 2 if i % 2 == 0 else i * 3 for i in range(0,8) ]
```

This code outputs list [2, 3, 4, 9, 6, 15, 8, 21] by evaluating the same conditional operation on each input list entry. For our core map pattern, we assume that each item is a scalar value. For example, if the map is over a tree, we assume the operation on a node does not access node parents nor node children.

Our pattern exploits that the instructions executed for an item can be predicted by whether the item is odd or even. Likewise, for webpage layout, the instructions for a map over the tree of document nodes can be predicted from the node type, such as Paragraph or List.

Impacting forces. Our map pattern is a relaxation of the traditional vectorizable branch-free `for-loop` and therefore presumes many similar conditions. For example, rather than operating over an array of structures, we assume data is already split into arrays [6].

Our pattern is sensitive to several factors. The cost of clustering similar nodes should not outweigh the benefit of clustered evaluation. Furthermore, our pattern permutes the order of the collection, which may impact

downstream computations. Finally, the amount of exploitable parallelism is determined by how many tasks with the same branch behavior can be grouped.

Solution structure. We split the task being mapped and the collection based on predictable branches. For the Python example, CDP evaluation would be:

```
1 clusteredData =
2   {'even': [ i for i in range(0,4) if i % 2 == 0 ],
3    'odd': [ i for i in range(0,4) if i % 2 == 1 ] }
4 result =
5   (@vector [ i + 2 for i in clusteredData.even ],
6    @vector [ i * 3 for in clusteredData.odd ])
```

By rewriting the loop into even and odd partitions, traditional loop vectorization can be applied to individual clusters (lines 5 and 6). In this case, the computation to perform is so small that the clustered data should be precomputed to see any vectorization speedups.

Generalization. The transformation can be understood as a variant of loop fission and conditional hoisting. The hoisted conditionals are now cluster conditions: a loop is executed for each branch body and only performed on data that matches the condition.

Our pattern supports operations on different types of collections. For example, it generalizes to topological traversals over trees. In a top down parallel traversal, the nodes of a level can be computed in any order, so they can be clustered. As another generalization, branch conditions can be nested or indirect. For example, if each node is a structured object with several fields, a nested conditional might run different branches based on different node field combinations. Branching may also occur due to method dispatch, such as the rotate example in Section 1. Our CSS case study (Section 4.2) exercises all of these variations.

2.3 Reductions: load balanced clustering

Parallel iterations over graphs often *reduce* the neighbors of each node. When the neighborhood size is small, SIMD instructions can still be used by simultaneously computing multiple reductions in lockstep, similar to segmented scans. [9] The reduction pattern is distinct from the previous branching map pattern. Reductions need not branch. Likewise, for maps, nodes need not communicate with their neighbors.

Clustering solves the problem of load imbalance. If simultaneous reductions have different lengths, the SIMD unit is occupied until the longest reduction completes. We balance completion times by running reductions in same-length clusters. Furthermore, to prevent domination by relayout costs (swizzling) induced by a naïve clustering, we present a recursive scheme.

```
1 for level in tree.levels.reverse():
2   @parallel for n in level:
3     if node.type == MAX_TYPE:
4       node.value = max([c.value for c in node.children])
5     elif node.type == MIN_TYPE:
6       node.value = min([c.value for c in node.children])
```

Figure 3: Naïve node-labeled reduction

Context. Compilers and XML processors reduce over trees. They invoke *visitors* that traverse trees and compute attribute values for nodes as they are encountered. For example, webpage layout computes the sizes of nodes in one pass and their dimensions in another.

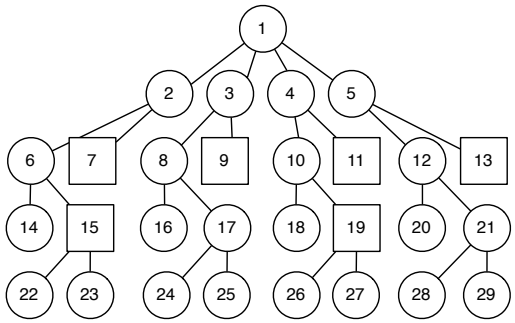
Reductions access the neighborhood of each node. For example, consider the *node-labeled reduction* over a tree in Figure 3 where leaf nodes have integral values and intermediate nodes are flagged as MAX_TYPE or MIN_TYPE. The reduction computes the value for each node based on the flag and the values of its children.

We address the challenge that there may be few children per node. To exploit parallelism, one node value can be computed simultaneously with the values for others on the same tree level. We optimize the choice by clustering based on neighborhood size. For example, a shopping cart on a webpage has a similar subtree for each item, so we exploit that the operations on one subtree are identical to operations on another.

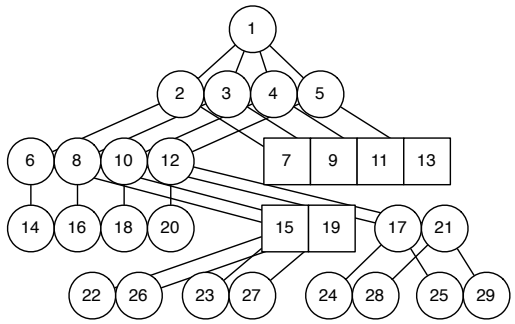
Impacting forces. We require many nodes to be on the same level. Furthermore, the benefit from the reduction pattern is proportional to the amount of computation spent in computing over the neighborhood for each node. Finally, irregularities such as instruction branching may require composition with our other patterns.

Solution structure. We observe that clustering nodes based on the number of children load balances simultaneous evaluation of multiple reductions in a cluster.

As an example of load imbalance, consider SIMD reductions over the breadth-first layout in the style of Chatterjee et al. [9]. Reducing over the children of nodes [6,7,8,9,10,11,12,13] in Figure 4 (a) runs two SIMD pairwise max instructions, one for each span ([6,7,8,9] and [10,11,12,13]):



(a) Breadth-first layout



(b) Nested clustering

Figure 4: Clustering for the reduce pattern. Before (a) and after (b). Shape denotes MIN_TYPE or MAX_TYPE and labels denote logical ID. Adjacent nodes are in the same cluster.

```
1 (value[6],value[7],value[8],value[9]) =
2   max4( [ value[14],NOOP,value[16],NOOP ],
3         [ value[15],NOOP,value[17],NOOP ] )
4 (value[10],value[11],value[12],value[13]) =
5   max4( [ value[18],NOOP,value[20],NOOP ],
6         [ value[19],NOOP,value[21],NOOP ] )
```

For each node of a span, SIMD evaluation will simultaneously compute over the same child index as for any other node in the same span. For example, the left operand is the first child of each span member and the second operand is the second children. If any of the nodes had a third child, we would have to add two more max statements, one for the third children of each span. Our imbalance problem is that, because the reductions have different lengths, there are many NOOPs.

Our initial solution in Figure 4 (b) clusters nodes based on the number of their children. The clusters are [6,8,10,12] and [7,9,11,13], requiring only the following single max4 under a naïve breadth-first data layout:

```
1 (value[6],value[8],value[10],value[12]) =
2   max4( [ value[14],value[16],value[18],value[20] ],
3         [ value[15],value[19],value[17],value[21] ] )
```

Calls of max4 are only for cluster [6,8,10,12]. There are no calls for [7,9,11,13], whose reduction lengths are 0.

The breadth-first layout in Figure 4 (a) is too costly if too few instructions are computed for each node. For example, loading values for nodes [14,16,18,20] is an inefficient *gather* of out-of-order data, which may dominate the computation. A similar problem occurs if we naïvely cluster each level of the tree by the number of children of each node.

Our final solution is to *recursively* cluster the nodes and layout the data as in Figure 4 (b). For example, nodes [2,3,4,5] are clustered, and then, recursively, so are their children. The children are colocated by their sibling number. For example, nodes [6,8,10,12] are colocated as the first children of cluster [2,3,4,5], as are [7, 9, 11, 13] because they are the second children. As a result, the i^{th} children of a cluster can be directly loaded for the i^{th} SIMD step of a reduction and the result of a reduction can be directly stored. For example, logical nodes [6,8,10,12] are now computed as:

```
1 (value[5],value[6],value[7],value[8]) =
2   max4( [ value[13],value[14],value[15],value[16] ],
3         [ value[17],value[18],value[19],value[20] ])
```

Clustering for load balanced SIMD evaluation introduced a data locality inefficiency not present in the unbalanced version: recursive clustering eliminates it.

Generalization. Our example supported the conditional in the reduction by composing with the map pattern. We included node.type as part of the clustering condition, so each cluster is guaranteed to branch in the same direction. For example, nodes [15,19] and [17,21] of Figure 4 (b) benefit from this clustering.

Different data structures such as graphs can be used as is described in the generalization of Section 2.1.

3. Analysis

In this section, we analyze the performance of our patterns, how to generalize them, and how to compose them. We found two basic properties to be central to our reasoning. First, the *compression ratio* measures the ability to cluster work. Second, the *clustering condition* is the invariant a cluster holds, which in turn impacts the speedup achieved in evaluating a single cluster.

3.1 Analyzing map, reduce, and search

CDP speedup can be analyzed in terms of properties r and T_c. We define r as the *compression ratio*: the number of clusters evaluated over the amount of sequential

work. We define T_c to be the optimized time to evaluate a cluster of nodes that satisfy the clustering condition. Total speedup is therefore $1 \,/\, r\, T_c$.

For example, sequential evaluation of the map in Section 2.2 has $r = 1$ (singleton clusters) and $T_c = 1$ (constant work per node). As expected, total speedup is 1 (no speedup). Seperate clustered evaluation of odds and evens using length p vectors has $r = 2/n$ and $T_c = (n/2)/p$, so the total speedup is p.

Our reduction pattern behaves similar as the map pattern. As with segmented scans [9] and our maps, speedup is proportional to cluster length rather than the number of children for a node. Important for commodity hardware, we do not require typically unavailable scan primitives. Finally, we do not expect small cluster sizes: the pigeonhole principle applied to large trees with a low branch count guarantees large clusters.

We also analyze binary search in terms of r and T_c:

$$r = \frac{1}{p} \tag{1}$$

$$T_c = 1 + \Pr[\text{miss}]p = 1 + (1 - b^p)p \tag{2}$$

$$speedup = \frac{1}{r\, T_c} = \frac{p}{1 + p(1 - b^p)} \tag{3}$$

The intuition for r is that our algorithm splits the search down the tree into clusters of p levels. The intuition for T_c is that the time to compute over a cluster is the cost of a correct speculation (1) and, in case of a misspeculation (probability $1 - b^p$), the time to compute the cluster sequentially (p). For example, if the speculation hit rate is high ($b = 1$), the speedup is p. Likewise, there is no speedup if the hit rate is low ($b = 0$).

In contrast, the FAST vector binary search algorithm of Kim et al. [16] will speculatively perform all of the comparisons for a subtree in parallel, not just for those on a hot path. The time to compute over a cluster of p levels therefore drops to $T_c = p/\log p$. The total speed of FAST is only $\log p$, compared to p for our algorithm when b approaches 1.

CDP algorithms vary in r and T_c. For example, much of our effort in the CSS case study was in lowering r. Highlighting the importance of T_c, our binary search scales linearly rather than logarithmically on hot paths.

3.2 Relayout Time

The time spent clustering is important. However, we did not find it to be a key concern for our case studies.

We use a simple two-pass algorithm for the map and reduce clusterings. The first pass determines the clusters and the second scatters the data. We also optimized finding a node's map cluster by hashing object fields. Both clusterings are embarrassingly parallel.

Clustering hot paths is more complicated. Chilimbi et al. [10] pioneered clustering techniques for hot path locality that might be adapted for JIT vectorization.

3.3 Clustering conditions for loop optimizations

While our focus is effective vectorization, there are other applications of clustering. Generalizing, we split loops into an outer loop and an inner loop by clustering the loop interval. The clustering guarantees a strong invariant that makes the inner loop optimizable (T_c). As sample sequential applications, we reused the map pattern to improve branch prediction and to hoist computations to occur between the inner and outer loops.

3.4 Compressing and composing clusterings

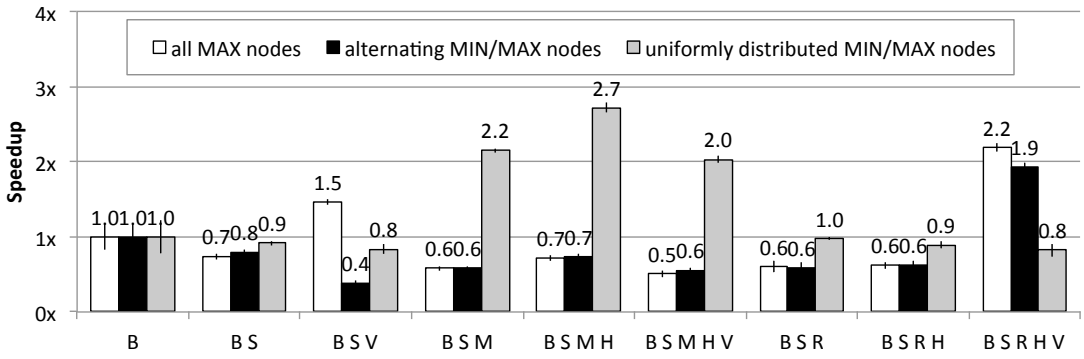
Pattern composition can be thought of as combining clustering conditions. However, we must not overly weaken the compression ratio when doing so.

Consider again the node-labeled reduction in Figure 3. We could partition based on node.type, enabling the map pattern, or on len(node.children), enabling the reduce pattern. The choice between node types is small, so the r for the map pattern is good, though the small amount of compute per node makes the T_c of the reduce pattern attractive. Our solution is to exploit both patterns: we check both of the clustering conditions for each node. The trade-off is that, r is, *at best*, the worse of the original compression ratios.

We can compose patterns without ruining the compression ratio. Of note, we implemented a composition operator that switches between clusterings. It partitions data into blocks and precomputes different clusterings for each block, one per clustering condition. Evaluation locally permutes a block depending on which clustering is needed. The compression ratio is improved at the cost of dynamic permute instructions added to T_c. Apple and IBM's PowerPC processors provide these local SIMD permute primitives, but software emulation of them dominated our experiments on Intel hardware.

4. Experiments and Case Studies

We manually rewrote several programs to follow our clustering patterns and call subword-SIMD compiler intrinsics. On map and reduce microbenchmarks, we see 2-3X speedups when using SSE4.2. For CSS webpage layout, we see 2-6X speedups and 2-8X perfor-



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

Figure 5: Map and combined map+reduce speedup over a breadth-first traversal on a 12 level tree. Error bars show 95% confidence of standard error over 40 cold-cache trials. Run with GCC 4.5.3 (-msse4.2 -O3) on a 2.6GHz Intel Core i7.

mance/Watt increases by using SSE4.2. Finally, targeting AVX, our binary search algorithm is 5.5X faster than FAST [16], the previously fastest algorithm, and has a 6.9X performance/Watt increase over it. [1]

4.1 Map and reduce microbenchmarks

Figure 5 shows 2-3X speedups for the MIN/MAX computation of Section 2.3 on a binary tree by manually inserting a clustering step and clustering the loops. Recall that a node's type determines whether to run a min or max operation, suggesting the map pattern. Likewise, there are few children per node, suggesting the reduce pattern. Both sequential and SIMD speedups depend on the distribution of MIN and MAX nodes.

Clustering improves sequential performance. As expected, the map pattern (B S M vs. B S) improves speedup for a random distribution of MIN/MAX nodes. Sequential performance slows down on regular trees (all-MAX nodes and alternating MIN/MAX nodes): clustering incurs the cost of irregular data accesses and cannot improve upon already predictable branching.

Similar reasoning applies to the reduce pattern (B S R) for regular trees. The recursive clustering of reductions is ineffective on random trees, however: a random distribution prevents large multi-level clusters.

Finally, hoisting the conditional out of each cluster (B S M H and B S R H) also improves speedup.

Clustered vectorization has the largest speedup. For regular trees, the reduce pattern yields a 2X speedup and strong scaling is 3.5X out of the ideal 4X (SSE4.2).

[1] We measure performance/Watt by reading energy performance counters after running as many trials as possible over 1 second.

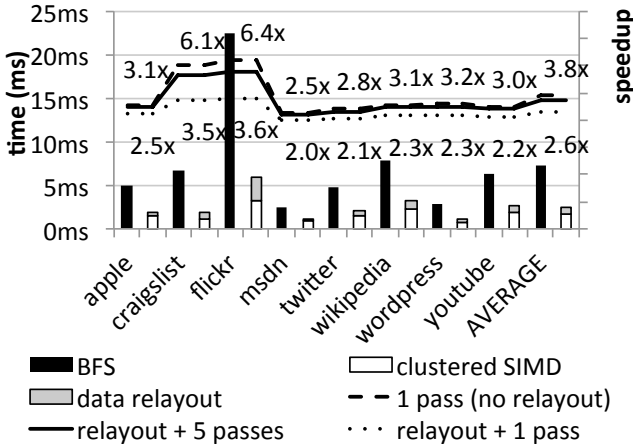


Figure 7: Impact of data relayout time on total CSS speedup. Bars depict layout pass times. Speedup lines show the impact of including clustering preprocessing time.

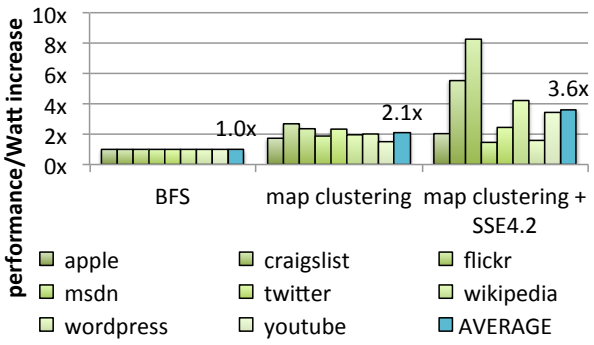
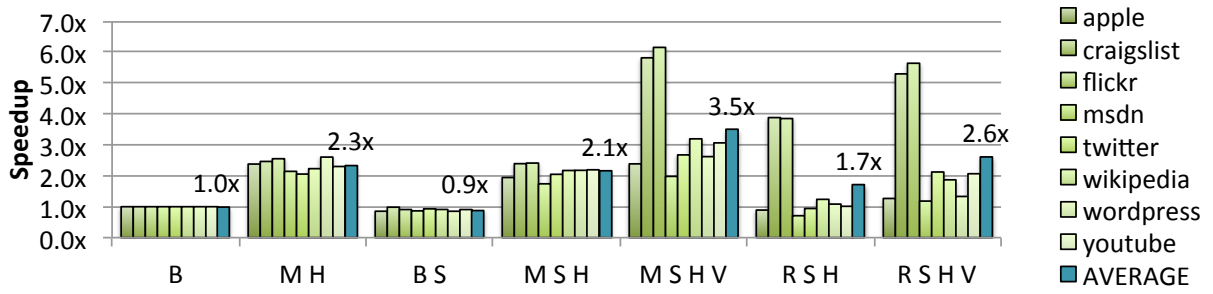


Figure 8: Performance/Watt increase for clustered webpage layout.



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

Figure 6: 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.

4.2 CSS webpage layout case study

We manually rewrote part of a webpage layout engine for 2-6X speedups and 2-8X performance/Watt increases. We found optimizing the compression ratio to be important. Finally, a significant 2.0-2.5X speedup was from sequential clustering optimizations.

Webpage layout is a sequence of tree traversals. A node visit will dispatch based on node type, such as paragraph or list, and branch on style fields, such as having a margin. This suggests the map pattern. Node values are also often a function of nearby nodes, such as the height being the sum of children heights. This suggests the reduce pattern. We report results from rewriting the widths traversal of the C3 web browser [18] and running popular pages (YouTube, WordPress, Wikipedia, Twitter, Flickr, Craigslist, etc.).

Clustering improves sequential performance. Applying map clustering (M H) yields an average 2.3X speedup because of branch prediction and hoisting. The recursive clustering (R S H) also yields up to 3.9X sequential speedups due to ordered child data accesses, but only when the compression ratio is high.

Clustering enables vectorization. Without it, we saw no SIMD speedups. Map vectorization (M S H V) gives across-the-board speedups with an average of 3.5X. Vectorization more than compensates for the sequential slowdown of structure splitting (M S H).

The map pattern clustering outperforms the reduce pattern clustering. Our intuition is two-fold. First, most of the computation of a node is typically not spent reducing, giving only a marginal benefit to clustered evaluation (T_c) under the reduce pattern. Second, the reduce pattern has a much worse compression ratio ($r = 40\%$) than the map pattern ($r = 12\%$), which clusters a level independently of other levels.

We experimented with which fields to cluster on for the map pattern. At first, we used a system-provided ID that already canonicalized many style attributes for a node, but then we switched to inspecting actual field values. Our compression ratio dropped from 78% to 12%, which correspondingly improved speedup.

Clustering time lowers speedup. Including clustering time for one traversal drops speedup to 2.6x. The cost can be amortized across multiple subsequent traversals; we estimate a 3.4x speedup for five passses. Optimizing the clusterer would also lead to speedups.

4.3 Binary search case study

We implemented our binary search algorithm using AVX intrinsics. Beyond the scope of this paper, we also achieved speedups for a similar case study of vectorizing the decision diagrams in a "web-scale" recommendation system by again clustering hot paths.

The performance of our binary search algorithm depends on the speculation hit rate. When the hit rate is over 40%, our algorithm is faster than FAST [16], the previously fastest algorithm. For perfect speculations, speedup is 5.5X over FAST with a performance/Watt increase of 6.9X.

We compare our algorithm to our best-effort reimplementation of the on-chip optimizations of FAST [16]. Both algorithms are speculative: they perform work that might not be needed in a sequential search. Our version is *optimistic*. Correct speculations follow a depth first traversal, while misspeculations must backtrack. In contrast, FAST is *eager*: it explores both branches from unrolling a loop iteration. FAST suffers from an exponential blowup when repeatedly un-

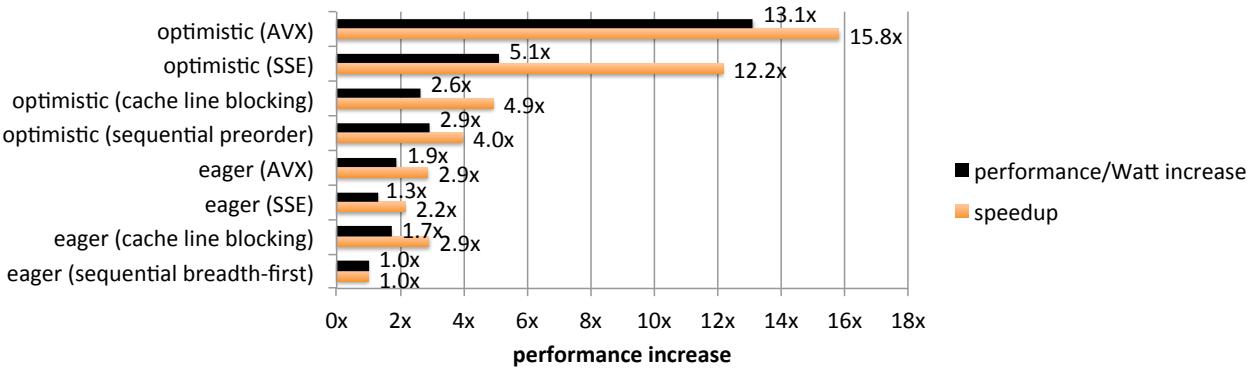


Figure 9: Binary search performance under perfect speculation. Baseline is sequential search on a breadth-first layout.

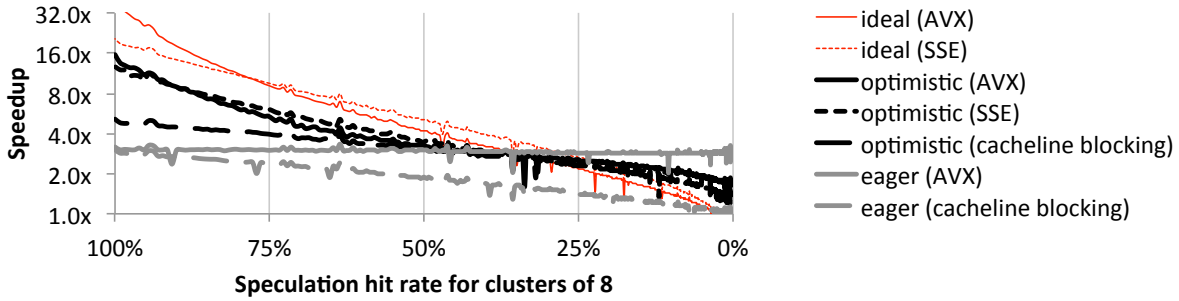


Figure 10: Speedup for clustered binary search as a function of the speculation hit rate.

rolling, preventing strong scaling. The degenerate case of a vector length of 1 causes FAST to be the traditional binary search over a breadth-first layout and for our algorithm to be the same over a preorder layout.

We run both algorithms using the Intel C++ Compiler 12.1 with flags `–std=c++0x –mavx –O3` on a 3.3 GHz Intel Core i7. We are interested in just CPU optimization, so we run on depth 17 trees.

Clustering improves sequential performance. Under perfect speculation (Figure 9), our *sequential* algorithm outperforms vectorized FAST by 1.4X.

Clustering improves vectorization. We see 2.5X and 1.3X relative speedups from adding SSE and AVX instructions. The corresponding relative improvement in performance/Watt is 2.0X and 2.6X. In total, we improve speedup over AVX FAST by 5.5X and performance/Watt by 6.9X.

Figure 10 shows the impact of misspeculation in terms of the likeliness of predicting 8 consective values. SSE instructions only require 4 consecutive predictions, so there should be a cross-over point at which SSE evaluation outperforms AVX. Based on our sequential performance, we expect the cross-over point to be at a 78% hit rate, but we actually observe it at 92%. Finally, we note that the cross-over point for switching to FAST is around 40%.

5. Related work

Loop transformations for vectorization are well-known [1, 3, 28]. CDP enables vectorization when the full loop cannot be vectorized. The closest techniques we build upon are for transforming data layouts for locality and for vectorizing nested data parallel programs.

5.1 Data layout optimizations

Chilimbi et al. [10] organize a tree in memory to better exploit caches. Given a fixed access pattern, they *collocate* data, meaning data is placed to improve locality by match spatial and temporal access patterns. Irigoin and Triolet [13], Frigo et al. [12], Ding and Kennedy [11], and Jo and Kulkarni [14] do as well, but further optimize the data access order, and target different sized caches. Our focus is instead on vectorization.

Similar to CDP's recursive use of striding for tree reductions, Nuzman et al. [24] apply data layout transformations to vectorization. Many such optimizations are known, e.g., structure conversion. We show how to target the map, reduce and graph traversal patterns.

5.2 Tree Pattern Optimizations

Designing vector algorithms is difficult. Indicative of this challenge, Barnes [4] provides an early algorithm in a paper titled "A modified tree code: Don't laugh; It runs". He targets a pattern not examined in this paper: concurrent queries in an all-to-all n-body simulation. Likewise, our binary search baseline is the manually implemented vector search by Kim et al. [16].

Matsuzaki et al. [19] study parallel tree reductions. Despite presenting an implementation, they do not report any speedups. Matsuzaki et al. [20] further discuss rose trees, finding data parallelism is effective for large trees over multiple processors. Our map, reduce, and traversal patterns can likewise be thought of as skeletons (operations on abstract types). We show similar speedups to their reduce example but on smaller trees and *only using one single-core processor*.

Notably, Chatterjee et al. [9] vectorize nested reductions by using scan primitives. CDP was motivated by the insufficiency of such nested data parallel algorithms for our case studies.

Our CSS case study is inspired by Meyerovich et al. [22]. They show that webpage layout is data parallel over a tree, but only explore MIMD optimization.

5.3 Language support

Many languages and frameworks automate vectorization. Automation is a natural next step for our algorithms and these systems feature relevant support for irregular computations.

Reps proposes vectorizing tree reductions with *scan grammars* [27]. Blelloch's NESL [6] achieves vectorization of structures with a bounded depth. Keller et al. generalize NESL to recursive types (e.g., trees of arbitrary depth) [15], which Data Parallel Haskell [26] employs. Unfortunately, the irregularity challenging scan grammars – nodes of different types – has not been addressed until our work. Traversals and unbalanced reduce are also unaddressed.

Data dependencies are visible to just-in-time compilers. Systems such as Copperhead [7], SEJITS specializers [8], and Intel Array Building Blocks [23] vectorize at runtime in order to exploit such statically unavailable knowledge. These techniques might be used to deploy our patterns.

Recently, Zhang et al. [29] showed that permuting arrays improves memory access patterns for irregular GPU programs. Our work is in a similar spirit, though our patterns focus more on data-dependent access patterns and control.

6. Conclusion

We have shown concrete vector algorithms for webpage layout and binary search, achieving a 4.1x and 4.9x over already optimized implementations.

We achieved these results by targeting their use of CDP patterns. In particular, we showed how to vectorize maps when there is branching, reductions when there are few elements per reduction, traversals when there is uncertainty about the traversal, and even combinations of these patterns. Our insight is that, instead of vectorizing an entire loop, we find clusters that are more amenable to optimization. Clustering is not a black art: we drew inspiration for our clusterings from data layout optimizations that targeted locality.

We believe clustering exposes new opportunities. Our binary search results suggest that there are opportunities in vectorizing basic algorithms. Larger programs may benefit as well: we are implementing a compiler support that applies our patterns as loop transformations. Finally, our evaluation is on restrictive subword-SIMD architectures, so we believe even better performance is possible on more relaxed SIMD architectures [17]. Considering our results, we believe there is significant future potential.

Acknowledgments

Herman Venter, Wolfram Schulte, Rastislav Bodik, Jim Larus, Andrew Gearhart, Lubomir Litchev, Bryan Catanzaro, Christopher Batten, Krste Asanavic, Dan Grossman, Nikolai Tillmann, and anonymous reviewers provided valuable guidance through various phases of this project.

Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship.

References

[1] Randy Allen and Ken Kennedy. Automatic translation of Fortran programs to vector form. *TOPLAS*, 1987.

[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, E. Lee, N. Morgan, G. Necula, D. Patterson, et al. The Parallel Computing Laboratory at UC Berkeley: A research agenda based on the Berkeley view. *EECS Department, University of California, Berkeley, Tech. Rep*, 2008.

[3] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*.

[4] Joshua E. Barnes. A modified tree code: don't laugh; it runs. *Journal of Computational Physics*, 1990.

[5] R. Bayer and E.M. McCreight. Organization and maintenance of large ordered indexes. *Acta informatica*, 1972.

[6] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 1994.

[7] Bryan C. Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *PPOPP*, 2011.

[8] Kamil Shoaib Lee Y. Asanovic K. Demmel J. Keutzer K. Shalf J. Yelick K. Catanzaro, B. and Fox A. SEJITS: Getting productivity and performance with selective embedded JIT specialization. In *First Workshop on Programmable Models for Emerging Architecture*, 2009.

[9] S. Chatterjee, G.E. Blelloch, and M. Zagha. Scan primitives for vector computers. 1990.

[10] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *PLDI*, 1999.

[11] Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *PLDI*, 1999.

[12] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. 1999.

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

[14] Youngjoon Jo and Milind Kulkarni. Enhancing locality for recursive traversals of recursive structures. In *OOPSLA*, 2011.

[15] Gabriele Keller and Manuel M. T. Chakravarty. Flattening trees. In *Euro-Par*, 1998.

[16] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*, 2010.

[17] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanovic. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *ISCA*, 2011.

[18] Benjamin S. Lerner, Brian Burg, Herman Venter, and Wolfram Schulte. C3: an experimental, extensible, reconfigurable platform for HTML-based applications. In *WebApps*, 2011.

[19] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Towards automatic parallelization of tree reductions in dynamic programming. In *SPAA*, 2006.

[20] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Parallel skeletons for manipulating general trees. *Parallel Computing*, 2006.

[21] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004.

[22] Leo A. Meyerovich and Rastislav Bodík. Fast and parallel webpage layout. In *WWW*, 2010.

[23] Chris J. Newburn, Byoungro So, Zhenying Liu, Michael D. McCool, Anwar M. Ghuloum, Stefanus Du Toit, Zhi-Gang Wang, Zhaohui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang. Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In *CGO*, 2011.

[24] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for SIMD. In *PLDI*, 2006.

[25] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. Alphasort: A cache-sensitive parallel external sort. In *VLDB*, 1995.

[26] Simon Peyton Jones. Harnessing the multicores: Nested data parallelism in Haskell. In *APLAS*, 2008.

[27] Thomas Reps. Scan grammars: parallel attribute evaluation via data-parallelism. SPAA, 1993.

[28] J. E. Smith, Greg Faanes, and Rabin Sugumar. Vector instruction set support for conditional operations. In *ISCA*, 2000.

[29] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In *ASPLOS*, 2011.

```
__m128i twos = _mm_set1_epi32(2);
for (int i = 0; < evenLen; i+=4) {
  __m128i even4 = _mm_load_si128(even + i);
  __m128i sum4 = _mm_add_epi32(twos, even4);
  _mm_store_si128(result + i, sum4);
}
... /* similar for odds */ ...
```

Figure 12: Map pattern: vector evaluation of an array of numbers clustered by even/odd.

```
for (int li = tree.numLevels − 1; li > 0; li−−) {
 Level *lvl = tree.level[li];
 int *val = lvl−>val;
 int *nextVal = tree.level[li + 1]−>val;
 for (int clstr = 0; clstr < lvl−>numClusters; clstr++) {
  int len = lvl−>clusterLen[clstr];
  int numChildren = lvl−>numChildren[clstr];
  if (lvl−>type[clstr] == MAX_TYPE) {
   for (int child = 0; child < numChildren; child++) {
    for (int n = 0; n < len; n += 4) {
     __m128i v4 = _mm_load_si128(al + n);
     __m128i cv4 = _mm_load_si128(nextVal + n);
     __m128i m4 = _mm_max_epi32(curVal4, childVal4);
     _mm_store_si128(val + n, m4);
    }
    val += len;
    nextVal += len;
   }
  } ... /* similar for MIN_TYPE */ ...
 }
}
```

Figure 13: Reduce pattern: recursively clustered vector evaluation of a MIN/MAX tree.

```
template<>
class Ops<float, __m256, 8> {
  public:
    static inline TVec loadPAligned(float *keys) {
      return _mm256_load_ps(keys); }
    static inline TVec gt(__m256 a, __m256 b) {
      return _mm256_cmp_ps(a,b,_CMP_GT_OS); }
    static inline int testAllOnes (__m256 m) {
      return _mm256_movemask_ps(m) == 255; }
};

template<typename T, class V, typename TVec, int VLen>
unsigned int searchNDFSIMD_PreorderBTree
(PreorderBTree<T, VLen> &tree, T val) {
  const TVec v = V::set(val); //splat
  const unsigned int bnodeSize =
    VLen == 1 ? 1 : (1 << VLen);
  const unsigned int fringeLen = 1 << VLen;
  unsigned int numSubtrees = tree.numBNodes;
  unsigned int i = 0;
  for (unsigned int bnodesBelow =
         VLen * ((tree.numLevels − 1) / VLen);
       bnodesBelow > 0;
       bnodesBelow −= VLen) {
    numSubtrees = (numSubtrees − 1)/fringeLen;
    const TVec span = V::loadPAligned(tree.data + i);
    const TVec cmp = V::gt(span, v);
    const int mask = V::testAllOnes(cmp);
    if (mask) {
      i += bnodeSize; //hot path
    } else {
      unsigned int iStart = i;
      int track = 0;
      for (unsigned int j = VLen; j > 0; j−−) {
        track = (track << 1) | (val > tree.data[i]);
        i += val < tree.data[i] ? 1 : (1 << (j − 1));
      }
      unsigned int nextBNode = track;
      i = iStart + bnodeSize
        + nextBNode * bnodeSize * numSubtrees;
    }
  }
  unsigned int lvlsLeft = VLen;
  while (tree.data[i] != val) {
    i += val < T(tree.data[i]) ? 1 : (1 << (lvlsLeft − 1));
    lvlsLeft−−;
  }
  return i;
}
```

Figure 14: Preorder binary search with abstracted vector instructions.

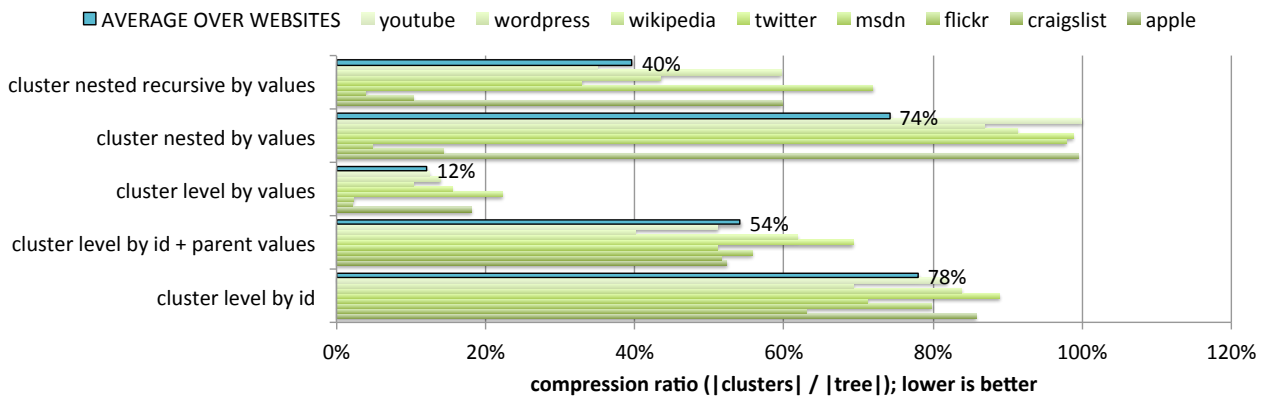


Figure 11: Compression ratio for different CSS clusterings. Bars depict compression ratio (number of clusters over number of nodes). Recursive clustering is for the reduce pattern, level-only for the map pattern. ID is an identifier set by the C3 browser for nodes sharing the same style parse information while value is by clustering on actual style field values.