Automatically Enhancing Locality for Tree Traversals with Traversal Splicing

Youngjoon Jo and Milind Kulkarni
School of Electrical and Computer Engineering
Purdue University
{yjo,milind}@purdue.edu

Abstract

Generally applicable techniques for improving *temporal* locality in irregular programs, which operate over pointer-based data structures such as trees and graphs, are scarce. Focusing on a subset of irregular programs, namely, *tree traversal* algorithms like Barnes-Hut and nearest neighbor, previous work has proposed *point blocking*, a technique analogous to loop tiling in regular programs, to improve locality. However point blocking is highly dependent on *point sorting*, a technique to reorder points so that consecutive points will have similar traversals. Performing this *a priori* sort requires an understanding of the semantics of the algorithm and hence highly application specific techniques.

In this work, we propose *traversal splicing*, a new, general, automatic locality optimization for irregular tree traversal codes, that is less sensitive to point order, and hence can deliver substantially better performance, even in the absence of semantic information. For six benchmark algorithms, we show that traversal splicing can deliver single-thread speedups of up to 9.147 (geometric mean: 3.095) over baseline implementations, and up to 4.752 (geometric mean: 2.079) over point-blocked implementations. Further, we show that in many cases, automatically applying traversal splicing to a baseline implementation yields performance that is *better* than carefully hand-optimized implementations.

Categories and Subject Descriptors D.3.4 [*Processors*]: [compilers,optimization]

General Terms Languages, Algorithms, Performance

Keywords locality transformations, temporal locality, cache, irregular programs, tree traversals

1. Introduction

Achieving high performance in many applications requires achieving good locality of reference. While there has been much work on automatic techniques for improving locality in *regular* programs, which operate over dense matrices and arrays [20], there has been comparatively little work on general techniques for improving locality in *irregular* programs, which operate over pointer-based data structures such as trees and graphs. There has been various work to enhance the *spatial* locality of pointer based structures, either at allocation time or through garbage collection, some of which are *automatic* [6–8, 22, 38, 40]. On the other hand, attempts at enhancing *temporal* locality has focused on *ad hoc* techniques that leverage application semantics [1, 2, 24, 28, 30, 32, 33, 36], or work well for sparse-matrix style algorithms [9, 27, 37], but not the tree- and graph-based algorithms that proliferate in domains like data mining and graphics. In this paper, we focus on *general transformation techniques to improve temporal locality in irregular programs.* If these techniques can be integrated into compilers, then we can offer efficient, locality-optimized implementations of irregular algorithms to programmers without requiring that they carefully hand-optimize their applications.

One important class of irregular applications is *traversal codes*, applications that perform repeated traversals of irregular data structures. Examples include well-known scientific algorithms such as Barnes-Hut [3], data mining algorithms such as point correlation and nearest neighbor [16], and graphics algorithms like ray tracing with bounding volume hierarchies [39]. These algorithms feature repeated traversals of highly irregular trees, with unpredictable application- and input-dependent traversal sizes, shapes and orders. Exploiting locality in these algorithms is critical because their performance is dominated by memory-access time, and careless accesses to irregular data structures are likely to result in cache misses. Any technique that can turn a substantial portion of those misses into hits has the potential to dramatically improve performance.

Prior work focusing on temporal locality in particular applications has proposed changing the order in which points (*i.e.* the entities that traverse the tree) are processed [2, 24, 28, 36]. By "sorting" the points such that points processed consecutively have similar traversals, locality can be enhanced (as discussed in Section 2). Unfortunately, performing point sorting requires analyzing the points prior to the traversals to rearrange them effectively. Determining an efficient way of performing this *a priori* sort requires an understanding of the semantics of the algorithm and hence highly application-specific techniques. Indeed, for some algorithms, the traversals are so complex that it is unclear how to do an *a priori* sort of the points to maximize traversal overlap, even when armed with semantic knowledge.

In prior work we discussed an abstract model of tree traversal codes that analogizes them to doubly-nested loops as seen in regular algorithms like vector outer product [19]. The outer loop is a loop of *points* that must traverse the tree, while the inner loop is a loop over the *nodes* that make up the traversal, irrespective of their position in the tree. Using this model, we proposed a transformation called *point blocking*, which essentially "tiles" the point loop: rather than performing a single point's entire traversal before moving on to the next point, a group of points are placed into a block, and the block traverses the tree, with each point in the block interacting with the necessary portions of the tree. Point blocking can deliver substantial locality and performance improvements for large traversals. Unfortunately, point blocking's performance is sensitive to the order in which points are processed; achieving the best possible performance from point blocking requires that the point sorting optimization be performed. As a result, point blocking is not a truly automatic, application-agnostic technique, depending instead on point sorting.

Given the difficulty of reasoning about the behavior or locality of irregular algorithms, it is unlikely that optimizations such as point sorting will be effectively applied to most implementations of tree traversal algorithms. In fact, since point sorting is *only* useful as a locality enhancement technique, it may not be applied at all by a non-locality-aware programmer. As a result, point blocking's effectiveness is muted in the typical case: when applied to naïve code that does not consider locality effects. Because our goal is to develop automatic compiler transformations to improve the locality of these algorithms, we need a transformation that does not assume any semantics-based, application-specific intervention by the programmer.

Our approach: traversal splicing

In this paper, we propose a new optimization, *traversal splicing*, introduced in Section 3. Much as point blocking tiles the point loop, traversal splicing tiles the *traversal* loop: each point's traversal is divided into a number of partial traversals, and we perform a partial traversal for all points before moving on to the next partial traversal for any point.

There are two key advantages to traversal splicing. First, the performance of splicing is largely independent of the order of points, decoupling its behavior from application-specific sorting transformations. Second, the order in which partial traversals are performed can be changed during execution. In particular, *as points traverse the tree*, we can use their traversal history as a predictor of their remaining traversal patterns, and group similar points together. In essence, traversal splicing *sorts the points on the fly* but *without any application-specific optimization*.

Section 4 discusses how to mitigate the overheads of traversal splicing by exploiting general structural properties of traversal algorithms. Section 5 describes an automatic, source-to-source transformation framework that can apply traversal splicing to any application that performs repeated, recursive traversals of a tree, and presents a tuning framework that automatically selects the appropriate optimization parameters for traversal splicing.

We evaluate our traversal splicing framework on six benchmark algorithms, and show that traversal splicing delivers (single-thread) speedups of up to 9.147 (geometric mean: 3.095) when compared to straightforward implementations of these algorithms, and up to 4.752 (geometric mean: 2.079) when compared to point-blocked versions. Furthermore, for each benchmark we compare a traversal-spliced implementation to a manually transformed version with both point sorting and point blocking applied. We find that across our benchmarks, *automatically applying traversal splicing to naïve implementations is competitive with hand-optimized implementations*.

Contributions

The contributions of this paper are:

- The development of *traversal splicing*, a new, general transformation for tree traversal codes which can effectively transform applications in the absence of semantics-based optimizations.
- The implementation of a transformation and tuning framework that can automatically transform tree traversal algorithms to apply traversal splicing.
- Experimental evidence that traversal splicing can not only effectively improve the performance of tree traversal codes, it can provide results competitive with hand-transformations that carefully leverage application semantics.

2. Background

2.1 Tree traversal algorithms and locality

The pattern of repeated tree traversals is a recurring theme, appearing in algorithms such as Barnes-Hut [3], nearest neighbor [16], iterative closest point [17] and many ray tracing algorithms [39], among others. We adopt some unifying terminology when discussing these algorithms: *points* are the entities that traverse the tree (they may be astral bodies in Barnes-Hut, rays in ray tracing, etc.), while *nodes* are the

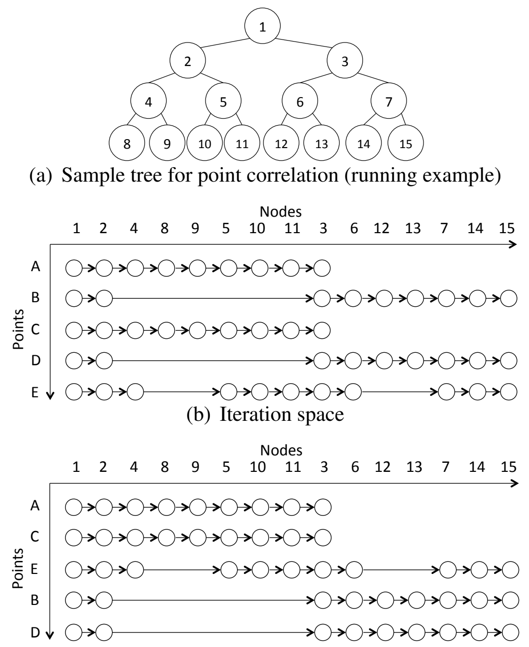
```
Set<Point> points = /* points */
Node root = buildTree(points);
foreach (Point p : points) {
  recurse(p, root);
}

void recurse(Point p, Node n) {
  if (!canCorrelate(p, n.boundingBox)) {
    return
  } else if (n.isLeaf()) {
    p.updateCorrelation(n.getPoint());
  } else {
    recurse(p, n.leftChild);
    recurse(p, n.rightChild);
  }
}
```

(a) Pseudocode of point correlation

```
Set<Point> points = /* points */
foreach (Point p : points) {
  foreach (Node n : p.oracleNodes()) {
    visit(p, n);
  }
}
```

(b) Point correlation as doubly nested loop

Figure 1. Point correlation



(a) Sample tree for point correlation (running example)

(b) Iteration space

(c) Iteration space, post-sorting

Figure 2. Sample tree and iteration spaces

individual elements of the tree data structures that are being traversed. Our definition of a tree traversal algorithm is thus: *an algorithm where each of a set of points recursively traverses a tree of nodes*. Note that the traversals in these algorithms are recursive, and hence depth-first.

To explain the behavior of tree traversal algorithms, we will use point correlation (PC) as an example. The two-point correlation can be calculated for a set of points by determining, for each point, p, the number of other points in the set that fall within a certain radius, r of p. PC is an important algorithm in many disciplines, such as bioinformatics and data mining [16].

The naïve approach to PC would be to compare each point to every other point in the data set, an $O(n^2)$ process. To accelerate the procedure, the standard approach is to build a spatial structure over the points called a *kd-tree* [4]. This structure is built top-down: a root node is created with a bounding box that encompasses all the points. Then a *split-plane* is computed that partitions the points in the bounding box into two equal pieces, creating two children nodes for the root, each with their own bounding box. This process is repeated until the leaf nodes contain single points. Now PC can be performed by a recursive traversal of the kd-tree. Each point p starts at the root and only traverses a child if the bounding box of that child can contain points within r of p. Thus, large portions of the tree need not be traversed, reducing the overall run time. The pseudocode for this algorithm is given in Figure 1(a).

While all tree traversal algorithms we consider have the same basic structure, they each traverse their trees according to different criteria and use trees with different structures (oct-trees for Barnes-Hut, kd-trees for nearest neighbor, bounding volume hierarchies for ray tracing), and dynamically allocate their trees according to input data. Hence, it appears at first glance that there may not be any unifying principles governing their locality. However, in prior work, we suggested eliding the traversal pattern, and indeed the tree structure itself, and instead considering each point's traversal as if the set of nodes it must visit were provided by some oracle [19]. This abstraction yields a consistent view of each algorithm as a simple doubly-nested loop, as shown in Figure 1(b) (the `visit` method abstracts whatever computations a point performs when it visits a node of the tree). As we shall see, this simple abstraction allows us to reason effectively about not only the locality behavior of a tree traversal algorithm, but also the effects of transformations applied to the algorithm.

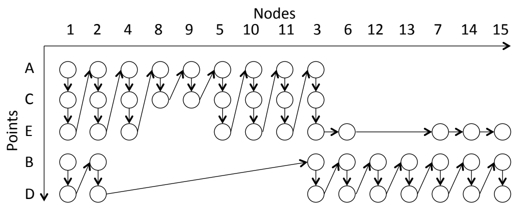
Figure 2(a) shows a sample kd-tree for PC, with nodes numbered in heap order, and figure 2(b) shows an iteration-space diagram for one set of traversals; this will serve as a running example throughout the paper. Each circle in the iteration space diagram represents one dynamic instance of the loop body. The vertical axis shows the outer loop over the points, while the horizontal axis shows which nodes are visited by each point. Note that each point does not visit each node. We can use reuse distance [25] to analyze the locality behavior of the algorithm. We note that each point enjoys good locality, while each tree node has relatively poor locality: concentrating on tree node ④, we see that between point A's first access to ④ and C's reuse of ④, we must visit all 14 other nodes of the tree before we return to ④. In general, the reuse distance for most nodes will be proportional to the size of the tree.

2.2 Point sorting

One popular approach to improving the locality of tree-traversal codes is to "sort" points so that points with similar

```
Set<Block<Point>> blocks = /* points */
foreach (Block<Point> b : blocks) {
  foreach (Node n : b.oracleNodes()) {
    foreach (Point p : b.validPointsAt(n) {
      visit(p, n);
    }
  }
}
```

(a) Abstract pseudocode for point blocking



(c) Iteration order for point blocking

Figure 3. Point blocking

Bench mark	Runtime (seconds)		CPI		L2 miss rate	
	Unsort	Sort	Unsort	Sort	Unsort	Sort
BH	102.7	27.9	2.44	1.23	0.429	0.021
PC	448.0	216.9	1.70	0.85	0.385	0.043
NN	383.8	190.2	2.00	1.13	0.486	0.227
kNN	529.3	147.8	2.43	0.91	0.481	0.226
BT	687.2	245.2	2.12	0.81	0.431	0.184
RT	155.5	102.5	1.31	1.10	0.275	0.163

Table 1. Efficacy of sorting with point blocking

traversals are executed close together, as in Figure 2(c), which shows the same traversals as Figure 2(b), but in a different order. Because points have different traversals, this sorting can reduce reuse distance; when point A and point C are executed consecutively, the reuse distance of tree node ④ drops to 8, and, in general, the reuse distance for a node will be proportional to the size of a *traversal*.

Unfortunately, *the right execution order for points is application-specific* and can require significant programmer effort to divine. Proposals that rely on point sorting adopt approaches such as using the tree-order of points [36] or space-filling curves [2]. Choosing the appropriate order for a given algorithm requires deep knowledge of the algorithm's behavior; this is especially problematic for algorithms such as nearest neighbor, where different points traverse the tree in different orders. In Section 3, we introduce a new locality optimization that performs this sorting "on-the-fly" and does not require any application-specific knowledge to implement.

2.3 Point blocking

Though sorting is a useful optimization that reduces the reuse distance for tree nodes to be on the order of traversal size, it loses its effectiveness when inputs, and hence traversal sizes, get too large. Prior work introduced *point blocking* as a method for improving locality for tree traversal codes when traversal sizes attenuate the benefits of sorting [19]. The essence of point blocking is to "tile" the point loop, yielding the abstract pseudocode shown in Figure 3(a). The arrows in Figure 3(b) show the new iteration order when applying point blocking to the sorted traversals of Figure 2(c), with block size 3. Note that the tree nodes enjoy improved locality: they will incur misses once per block, instead of once per point. Further, the reuse distance of a point is on the order of the block size, so as long as blocks are properly sized, points will suffer only cold misses.

We note, however, that point blocking's effectiveness relies on point sorting as a preprocessing pass. A more precise characterization of the locality behavior of a point blocked code is that a tree node suffers one miss *per block that visits it*. If the points are sorted, then points that visit a particular node are likely to be collected into a relatively small number of blocks, and hence the node will suffer few misses. If the points are unsorted, each block will have to visit more tree nodes (as its points' traversals will have less overlap and hence cover more ground). Thus, each tree node is visited by more blocks, and will suffer more misses. Consider node ④ from our running example. In the sorted version of the point-blocked code (Figure 3(c)), all the points that visit ④ are in the same block, and ④ only suffers a single miss. However, if we were to apply point blocking to the original order of points from Figure 2(b), we note that two blocks would have to visit ④, resulting in two misses.

Table 1 shows the runtimes, CPI and L2 miss rates of several benchmarks with point blocking, with and without sorting[1]. While point blocking can often improve performance for both scenarios with and without sorting, the gap between sorted point blocking and unsorted point blocking remains significant. Hence we would like to combine point blocking with point sorting to achieve the best performance. Unfortunately, as discussed before, performing sorting requires application-specific knowledge, and a general transformation cannot rely on having sorted inputs. In the next section, we introduce a transformation whose performance is less dependent on the order in which points are processed, obviating the need for application-specific sorting.

3. Traversal Splicing

This section introduces *traversal splicing*, a novel transformation for tree traversal algorithms that addresses the shortcomings of the techniques discussed in Section 2. In particular, it does not rely on any application-specific semantic knowledge (*e.g.*, how to sort points) to work effectively. In other words, traversal splicing can deliver good results even for simple, baseline implementations, without relying on programmer intervention to enhance locality.

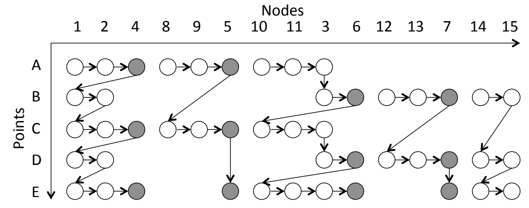
3.1 What is traversal splicing?

The most intuitive way to visualize traversal splicing is that, rather than tiling the point loop, as in point-blocking, it tiles the traversal loop, yielding the abstract pseudocode of Figure 4(a). Thus, rather than picking a block of points and following their traversals through the entire tree, traversal splic-

[1] The results in Table 1 are obtained via PAPI [10], with the random inputs on the Opteron II machine, described in more detail in Section 6.1.2.

```
1 Set<Point> points = /* points */
2 Set<Set<Node>> traversals = /* tree */
3 foreach (Set<Node> t : traversals) {
4   foreach (Point p : points) {
5     foreach (Node n : p.oracleNodesWithin(t) {
6       visit(p, n);
7     }
8   }
9 }
```

(a) Abstract pseudocode for traversal splicing



(b) Iteration order for traversal splicing

Figure 4. Traversal splicing

ing takes a single point and executes a *partial traversal* of the tree. It then takes the next point and executes a partial traversal and so on. Once each point has executed a partial traversal, the first point's traversal picks up from where it left off. This process can be extended by dividing each traversal up into several partial traversals, whose executions are interleaved. In essence, each point's traversal is chopped up into pieces, and the pieces are rearranged and stitched together in a different order; hence, *traversal splicing*. The nodes at which the traversals are paused are called *splice nodes*.

Figure 4(b) shows the effects that traversal splicing has on the iteration space from Figure 2(b), with the iterations that visit the splice nodes (④, ⑤, ⑥ and ⑦) filled in[2]. Note that the partial traversals are executed in "lock-step," and if a point does not encounter a splice node (consider point B, which does not visit node ④ or ⑤), it resumes once all other points have arrived at the next node it should visit.

We can use the iteration space diagram to reason about the locality effects of traversal splicing. We note that each node in the tree has good locality. The reuse distance of a tree node is bounded by the distance between splice nodes. As long as the splice nodes are not too far apart (*i.e.,* each set of nodes in line 3 of Figure 4(a) fits in cache), we get only cold misses on the tree nodes. The points, however, will miss once per partial traversal (when a point pauses at a splice node, it will not be re-accessed until every other point has completed a partial traversal). Interestingly, the order of points is irrelevant to this locality analysis. Hence, *traversal splicing is agnostic to whether or not the points are sorted.*

Dynamic sorting Note that the above analysis assumes that splice nodes are not too far apart. Unfortunately, the structure and size of the tree can be extremely irregular, leading to sets of nodes (line 3 of Figure 4(a)) that may exceed cache. However, the *partial traversals* of points (line 5 of Figure 4(a)) may still fit in cache. In this situation, the order of points once again matters: points with similar partial traversals between splice nodes can enjoy good locality if they are processed consecutively. But how can this reordering be done automatically?

To perform this scheduling, we exploit a key insight about tree traversal algorithms. Two points that reach the same splice node of a tree have had similar traversals up to this point (points with substantially dissimilar traversals will have been truncated prior to arriving at the splice node). Furthermore, their traversals' similarity implies that the continuations of the traversals are likely to be similar as well. We can thus use the order in which points reach splice nodes as a proxy for the similarities of their traversals. Hence, we will *reorder the points* as they arrive at splice nodes.

As an example, consider applying this strategy to the traversals of Figure 4(b). We will process the points in their original order until splice node ④. Because points B and D did not reach node ④, they will be reordered with respect to points A, C and E. The order in which the points will be processed for the partial traversals between ④ and ⑤ is (A, C, E, B, D). Note that this new order is precisely the order in which the points would have been executed had they been sorted *a priori* (see Figure 2(c)).

As the traversals continue, the points arrive at ⑤ in their current order, so no reordering is done. Next, the points will continue on to ⑥. Note that the reuse distance for ⑥ is improved: points B and D are now processed consecutively. At ⑥, the points will be reordered again, to (E, B, D, A, C), and so on. This continuous reordering has the effect of sorting the points as they traverse the tree, so that points with similar traversals will wind up near each other in the processing order.

Combining traversal splicing and point blocking Finally, if we consider the inner two loops of Figure 4(a), we note that it looks like the abstract version of the original traversal code. If the partial traversals become too large, it is clear that applying point blocking to the inner two loops (which, recall, are now operating over dynamically sorted points) can further enhance locality. Section 5.4 discusses how applying point blocking to partial traversals allows our transformed code to tolerate larger partial traversals, and hence reduces its sensitivity to splice node placement. Section 6 quantifies the performance of point blocking and traversal splicing individually, and the combination of the two.

3.2 More complex traversals

Matters are more complicated when traversals of points do not take the same path through the tree. In PC, each point traverses the tree in the same order, aside from truncations; there is a single global traversal order, and each point's traversal is a filtered subset of that order. Figure 5 shows the pseudocode for nearest neighbor (NN), in which the traversal order is not fixed. For example, at a given node,

[2] Note that this assumes that every point in the algorithm performs its depth-first traversal in the same order; we discuss the implications of algorithms where points can visit nodes in different orders later.

some points may visit the node's left child before its right, while other points visit its right child before its left.

This scenario is both a more complex challenge for traversal splicing and a more promising opportunity. In applications like PC, each point visits the splice nodes in the same order, though truncation may prevent it from reaching a particular splice node. In NN, points may visit splice nodes in different orders, even disregarding the effects of truncation. Because the traversals have the potential to diverge substantially, leaving the points unsorted can yield very poor performance. Further, the exact path of each traversal may be highly input dependent, making *a priori* sorting difficult.

In such a scenario, each point follows its prescribed traversal until it reaches its first splice node, even if different points reach different splice nodes. Splicing, with reordering, occurs as before. Then points continue on to the second splice node, and so on. Hence, each point's traversal is still divided into partial traversals, and the partial traversals are still executed in lockstep.

More precisely, the computation is divided into n *phases*, one per splice node (and hence one per partial traversal). In phase i, each point p executes the partial traversal starting at the $(i-1)$th splice node and ending with the ith splice node in that point's particular traversal. After each phase, the points at each splice node are sorted according to their traversal history as before. Note that this phasing approach is merely a generalization of the splicing procedure for applications like PC. In PC, because each point follows the same path through the tree, every point's ith phase ends at the same splice node.

The only complication is when a point is truncated before reaching a splice node, skipping one or more splice nodes. In this case, the truncated point conceptually still participates in that splice node's phase, but without performing any work; the point's traversal will resume when the phases for any skipped splice nodes are over. Section 5 describes this phasing algorithm, and how it can be scheduled efficiently, in more detail.

```
Set<Point> points = /* points */
Node root = buildTree(points);
foreach (Point p : points) {
  recurse(p, root);
}

void recurse(Point p, Node n) {
  if (!canBeCloser(p, n.boundingBox)) {
    return;
  } else if (n.isLeaf()) {
    p.updateClosest(n.getPoint());
  } else {
    double split = p.value(n.splitType);
    if (split <= n.splitValue) {
      recurse(p, n.leftChild);
      recurse(p, n.rightChild);
    } else {
      recurse(p, n.rightChild);
      recurse(p, n.leftChild);
    }
  }
}
```

Figure 5. Pseudocode of nearest neighbor

3.3 Correctness

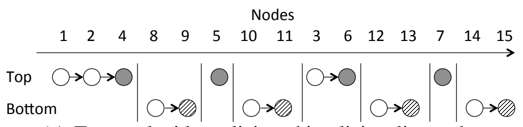
Traversal splicing performs a comprehensive restructuring of a program's accesses to a tree. Care must be taken, therefore, to ensure that the restructuring does not violate any dependences. To explain the criteria governing the correctness of traversal splicing, we appeal to the iteration space diagram, and draw an analogy between traversal splicing and loop tiling. Traversal splicing is the equivalent of "tiling" the traversal loop in the doubly-nested loop formulation of tree traversal algorithms shown in Figure 1(b), and the correctness criteria for loop tiling (that the loop be fully permutable) still apply. In particular, the order in which a point visits nodes in its traversal is unchanged, as is the order in which a given tree node is visited by different points. Thus, traversal splicing can be performed in the presence of intra-traversal dependences (*e.g.*, if some data associated with the point is updated at each leaf node of the point's traversal) or dependences that cross traversals but stay within the same node (*e.g.*, if a counter at each node is updated whenever a point visits the node).

When reordering is performed during splicing, the correctness criteria change. Each point's traversal still occurs in the prescribed order, and hence intra-traversal dependences are preserved. However, because the order in which partial traversals happen can be shuffled around, inter-traversal dependences are no longer guaranteed to be preserved. Nevertheless, *loop parallelizability* is a sufficient condition: if the outer point loop of the original traversal algorithm can be parallelized, traversal splicing is legal. This same condition is necessary for any application-specific, *ad hoc* point sorting optimization to be legal.

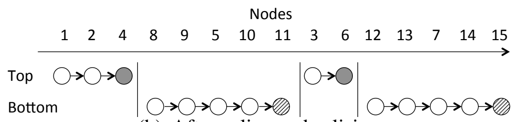
3.4 Splice node placement

In principle, splice nodes can be located at any point in the tree. Indeed, different points can use different splice nodes. However, there are certain principles that govern the selection of splice nodes.

1. If different points exhibit different traversal orders, the phasing algorithm outlined above requires that each point that will encounter a splice node in phase i encounter that phase's splice node at the same time. This can easily be accomplished by placing all splice nodes at a uniform depth from the root node . Section 4.4 discusses how this criterion can be relaxed.
2. The deeper in the tree splice nodes are placed, the more splice nodes, and hence partial traversals, there are. Placing splice nodes too deep can result in high overhead (due to the extra bookkeeping necessary to keep track of partial traversals) and poor locality (as points incur a miss once per partial traversal).
3. Conversely, if the points are too high in the tree, then too many points will reach the same splice nodes, reducing the efficacy of the reordering optimization. Further, the portions of the traversal "below" the splice nodes will be



(a) Traversal with explicit and implicit splice nodes



(b) After splice node elision

Figure 6. Top and bottom phases

large, potentially resulting in poor locality, though this effect can be mitigated by applying point blocking, as discussed in Section 3.1

Good splice node placement requires striking a balance between placing the nodes too shallow in the tree for reordering to be useful and placing them too deep to take advantage of reordering. Section 5.4 describes our approach to splice node selection.

4. Optimizations

Traversal splicing as described in Section 3 comes at a cost. It requires that traversals be paused and resumed at splice nodes. In principle this would require maintaining the full stack for each point's traversal, allowing it to be paused at a splice node and its continuation resumed in the next phase. Rather than storing a point's stack in some ancillary data structure, we record the information in the tree itself. When a point is at some node n in its traversal, each level of its stack can be stored in the appropriate ancestor of n. Hence, at each node in the tree, we will store, per point, any information needed by the point at that level in its recursion. This includes any local variables of the recursive method, as well as a program counter, recording where in the recursive method the point was when it descended from the node. This information needs to be tracked for each point, and can consume space proportional to the depth of any traversal. Because all points are in flight simultaneously, the amount of extra space required per point can lead to prohibitive overheads for traversal splicing.

This section discusses optimizations that take advantage of *structural characteristics* of the target tree traversal algorithm that allow traversal splicing to be implemented with far less space overhead. In the following presentation, D refers to the depth of the splice nodes (recall from Section 3.4 that we place all splice nodes at a uniform depth from the root).

4.1 Implicit splice nodes

To avoid storing stack information at *every* node along a point's traversal, we note that partial traversals that start after a splice node (*e.g.*, the partial traversals starting after node ⑤ in Figure 4(b)) visit the entire subtree of the splice node before returning higher in the tree. We introduce the concept of *implicit* splice nodes: nodes that act as splice nodes for the purposes of pausing and restarting traversals, but are not explicitly marked by the transformation. The last node visited by any partial traversal in the subtree "below" an explicit splice node is an implicit splice node. Figure 6(a) shows the resulting decomposition of a traversal over the tree in Figure 2(a). The explicit splice nodes are shaded in gray, while the implicit splice nodes are shaded with diagonal lines. We can categorize the partial traversals as "top" traversals that traverse the top portions of the tree and end at an explicit splice node, and "bottom" traversals that traverse the lower portions of the tree and end at an implicit splice node. Notably, the bottom traversals are equivalent to normal recursive traversals over the subtrees rooted at the explicit splice nodes. Therefore, we need only track information for traversal splicing for top phases. Because any top phase is no larger than a single path from the root to a splice node, the maximum amount of space required to track each point is now $O(D)$.

4.2 Reducing stack storage

With the addition of implicit splice nodes, a point's stack only needs to be explicitly tracked in top phases. We next aim to reduce the amount of state that needs to be saved during the top phases. We note that tail recursion optimization is commonly performed for recursive methods: if the last operation by a recursive method is a recursive call, then the stack does not need to be saved upon making the call. While the recursive methods in tree traversals tend not to be purely tail recursive (as there are recursive calls for each child node), we can consider *pseudo-tail recursive* methods. A *pseudo-tail recursive* method is a recursive method for traversing the tree where any recursive call within the method is immediately followed either by another recursive call or a method exit. Because no local variables are used between recursive calls, we need not track any information other than a program counter for each point at each level in the recursion.

To track a point's program counter efficiently, we group each straight-line series of calls in a pseudo-tail recursive method into a *call set*. Each call set has a fixed sequence of recursive calls (*i.e.*, a fixed order of visiting a node's children), so a point's behavior at this node is completely determined by which call set it uses, and the call set it uses is computed before the first recursive call.

The nearest neighbor (NN) code of Figure 5 shows an example of a pseudo-tail recursive method with multiple call sets. Here the two possible call sets (traversal orders) are {leftChild, rightChild} (lines 15 & 16), and {rightChild, leftChild} (lines 18 & 19), and which call set a point will take is decided before any recursive calls are made. Thus, at each level, a point need only track which call set it used, and where in the call set it was, to fully reconstruct its stack. The phased nature of the traversal splicing algorithm means that every point's location in their respective call sets will be aligned; at any given time, every point will be executing the first recursive call of their call set, or every point will be executing the second recursive call of their call set, etc. Hence, at each depth we can track a global "phase number" that maintains where in its call set each point is. A point only needs to maintain which call set it is using at a given level.

4.3 Inferred order

We can further reduce the amount of storage required during traversal splicing by noting that in many algorithms, the call sets of the recursive method follow a particular pattern. Specifically, each call set makes the same number of recursive calls, and in a given phase of the algorithm, each call set is operating on a different child. That is, there is no i such that the ith call of two call sets are performed on the same child. Hence, at a particular level, if we know which phase the algorithm is in (what i is) and we know which child node the point traversed during the phase, we can infer which call set the point used at this level, and so no longer need to record it. NN is an example of an algorithm from which order can be inferred. The two call sets each have two calls, and in the first phase, the points in the first call set visit leftChild, and the points in the second call set visit rightChild. In the second phase, we know that any point that visited leftChild must now visit rightChild, and vice versa.

Note further that because the data structure being traversed is a tree, knowing where a point is in the tree at any given time during execution uniquely determines the path from the root to that point. Hence at a given level we need not store which child the point visited, either. This eliminates the need to track any information per point aside from a global phase number per level (shared across all points) and the current tree node the point is at. This reduces storage needs to $O(1)$ per point. A special case of inferred order is when there is a single call set in a recursive method as in PC (Figure 1(a)).

4.4 Splice node elision

Given our policy of placing splice nodes at a fixed depth, certain top phases, which begin just above the splice depth, will necessarily be very short. For example, the partial traversal starting at node ⑤ is only one node long! To avoid the overhead of performing splicing when it is unlikely to be effective, we *elide* splice nodes for short phases. That is, if a partial traversal is likely to be short, we combine it with the following partial traversal. In practice, for top phases that begin a short distance above the splice depth D, we do not perform splicing and instead immediately begin a bottom phase, as shown in Figure 6(b). This can be both good and bad for locality. It is good as we incur fewer misses on each point (as we have fewer phases), but the combined bottom phase becomes larger, potentially outstepping cache. When splice node elision is combined with point blocking, the latter effect is mitigated. Section 5.4 discusses our strategy for splice node elision.

```
1 Set<Point> points = /* points */
2 Node root = buildTree(points);
3 mapTree(root);
4 allocBuffers();
5 foreach(Phase p : unrollRecursion()) {
6   topPhase(p.depth, p.phase);
7   bottomPhase();
8 }
```

Figure 7. High level view of traversal splicing

5. Implementation

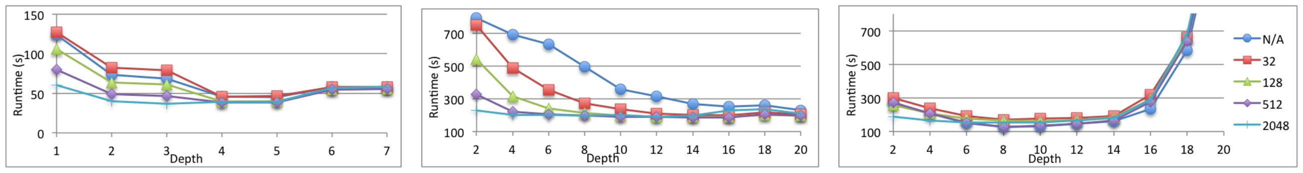
This section discusses how to realize the abstract concept of traversal splicing in actual code, how the "on the fly" sorting is implemented, and how splicing can be applied and tuned automatically. The splice depth is denoted D.

5.1 Splicing and sorting

Figure 7 shows a high level view of how traversal splicing is implemented. Recall that traversal splicing chops up a point's traversal into many partial traversals that are split at splice nodes, both explicit and implicit (Section 4.1). Bottom phases start at the children of the explicit splice nodes (nodes at depth $D{+}1$), but top phases start immediately after bottom phases complete. To find the nodes at which phases must begin, we map the top of the tree up to (and including) the children of splice nodes (line 3). Each of these nodes can be the start of a new top or bottom phase. For each of these nodes we allocate buffers into which points can be saved, so they can be resumed by top phases. We also allocate a buffer which can save $D + 1$ call set ids for each point (line 4).

The partial traversals are executed as pairs of top and bottom phases as described in Figure 6(a). Because a top phase can start at any node up to the splice depth, there are $2 \times p^D$ phases (where p is the maximum number of recursive calls within a call set, and 2 is for top and bottom each); `unrollRecursion` (described in Appendix A) identifies each phase. Each top phase (line 6) executes a *single* path through the tree to an explicit splice node, saving points and call set ids into the buffers, so that traversals can be resumed by later phases. Each bottom phase (line 7) executes *all* paths of the subtree rooted at an explicit splice node, *without* saving points and call set ids. The last node visited in the bottom phase is by definition an implicit splice node.

The first top phase starts every point at the root of the tree. Subsequent top phases and bottom phases gather points saved into buffers from previous top phases, and resume them at appropriate nodes based on the call set id. Bottom phases resume only points paused at explicit splice nodes, whereas top phases also resume points saved above splice nodes due to truncation. Hence top phases gather points from *multiple* nodes, and dynamic sorting (described in Section 3.1) arises naturally as points are reordered based on the node they were saved at. There is no dynamic sorting at bottom phases, as they gather points from a *single* node. More details of the implementation of both splicing and dynamic sorting are in Appendix A.



(a) Barnes-Hut (BH) (b) Point Correlation (PC) (c) Nearest Neighbor (NN)

Figure 8. Runtime with varying block sizes and splice depths

5.2 Local variables and intermediary methods

Because traversal splicing entails chopping up the recursion, it also requires that all methods surrounding the recursive method be split into prologues and epilogues. All prologues are executed before the first top phase, and all epilogues are executed after the last bottom phase. This requires that any local variables which are defined in the prologue(s) and used in the epilogue(s) be saved in additional space allocated per point. Further, any local variables within the recursive method passed as an argument to subsequent recursive calls must also be saved for all points at all depths. This can contribute significantly to heap usage as shown in Section 6.2.

5.3 Automatic transformation

We developed a transformation framework, called *TreeSplicer*, written as a series of passes in the JastAdd framework [11]. TreeSplicer can automatically apply traversal splicing, point blocking [19], or a combination of both. It identifies if splicing can be applied by examining if the recursive method is pseudo-tail recursive. TreeSplicer does not apply splicing to non-pseudo-tail-recursive codes. TreeSplicer determines which optimization level to use by producing a matrix of all call sets in the recursive method. If there are no conflicting traversals in any phase, the node inferred optimization can be used. We apply further optimizations when there is a *single* call set.

Recall that splicing is only performed on parallelizable traversals of trees. We currently rely on annotation to ensure that the recursive structure being traversed is a tree, and that there are no inter-point dependencies; determining these properties automatically is beyond the scope of this paper.

A more detailed discussion of TreeSplicer is deferred to Appendix B. We have made the source code of TreeSplicer public at https://sites.google.com/site/treesplicer.

5.4 Tuning the transformation

Critical to the performance of splicing is selecting a good splice depth, D. The parameters are dependent on both machine (*e.g.*, L1, L2 cache size) and algorithmic characteristics (*e.g.*, average traversal size). In our point blocking work, we proposed an autotuning technique for selecting block size where a small portion of the input points are used to construct blocks of varying sizes, and the best performing block size is chosen as the transformation parameter [19]. A similar approach is infeasible for splicing, because we would like to apply splicing over the full set of points.

In principle, we would like to choose the splice depth, D, such that each partial traversal completely fits in cache. In practice, because tree topology is unpredictable, and the size of partial traversals is input and algorithm dependent, finding a suitable D is difficult.

Recall that in Section 3.1, we observed that point blocking can be applied on top of traversal splicing to mitigate the effects of large partial traversals. Figure 8 shows the runtime of three benchmarks with varying block sizes and splice depths (on the Opteron system described in Section 6.1.2). The five lines compare splicing only, to splicing with blocking applied to the bottom phases[3] with block size 32–1024. We note that point blocking reduces traversal splicing's sensitivity to D for BH and PC, which have large bottom phases, and does not harm the performance of NN, which has small bottom phases. Serendipitously, this means that by combining splicing and blocking, we need not be as careful about choosing D. Empirical results suggest that a depth at half of the average *reach* (the depth of the nodes where a point's recursion is stopped) is a good splice depth.

We use the autotuning technique of prior work [19] to choose the block size, and record the average reach of all points in the test blocks. If we are not applying blocking, we use 10 randomly selected points to compute the reach. We set D as half of the average reach. We also determined empirically that splice node elision (see Section 4.4) should be performed if a top phase begins fewer than $D/2$ levels above D. As we limit the autotuning to less than 1% of the points, it consumes less than 1% of traversal time. The parameters chosen by the autotuner for our experiments are presented in Appendix C.

5.5 Discussion

Some algorithms dynamically generate new points. An example would be mirror rays in ray tracing, which we do not currently consider. One possible solution to handle this is to create an outer loop over the point set, and defer dynamically generated points to a subsequent outer-loop iteration. Very large number of points can require prohibitive amounts of memory, as the memory overhead of splicing increases proportionally to the number of points. For very large number of points it would be necessary to split the points into smaller groups, and apply splicing on a group at a time. We currently do not implement these features.

[3] We only apply blocking to bottom phases because top phases take a single path through the tree, and will be at most D nodes long.

6. Evaluation

This section presents our experimental evaluation. We start with evaluation methodology, then explore the memory overheads of splicing. Next we discuss experimental results for serial runs of each of our benchmarks, and show that splicing is often competitive with manual optimizations that exploit semantic knowledge. Finally we discuss parallel results and demonstrate that traversal splicing scales to many threads.

6.1 Evaluation methodology

To demonstrate the efficacy of TreeSplicer, we evaluate it on six tree traversal algorithms, from various domains ranging from scientific applications to data-mining and graphics[4]. We evaluate five versions of each benchmark.

- **Base**: the baseline described for each benchmark below.
- **Block**: automatic point blocking as described in prior work [19].
- **Splice**: automatic traversal splicing as described in Section 5.
- **Block+Splice**: automatic traversal splicing combined with automatic point blocking for bottom phases.
- **Block+SpliceElision**: the splice node elision optimization is added to **Block+Splice**.

Note that our baseline benchmarks are true baselines: no application-specific a priori *sorting is performed.* The benchmarks were written in Java and executed on the HotSpot VM 1.7 with 12GB heap. Each configuration was run 8 times in a single VM invocation, the first run and the min/max runs dropped, and the mean of 5 runs was recorded. We enforced a coefficient of variation[5] of 0.05 by extending the number of runs until steady state if necessary, and this yields errors of at most $\pm 6.21\%$ of the mean with 95% confidence [13].

6.1.1 Benchmarks

Barnes-Hut (BH) is a scientific kernel for performing n-body simulation [3]. All n bodies are placed into an oct-tree. Each body traverses the tree to compute the force(s) acting upon it. In the terminology of our optimization classes from Section 4, BH is an inferred order algorithm with a single call set. We use the implementation from the Lonestar benchmark suite [21] with a single iteration, and two inputs. **Random** is one million randomly generated bodies. **Plummer** is the class C input from the Lonestar suite, with one million bodies generated from a Plummer model.

Point Correlation (PC) is described in detail in Section 2. PC is an inferred order algorithm with a single call set. We use three inputs. **Random** has one million randomly generated points in 3 dimensions. **Covtype** is real data on forest cover type with $580{,}000$ points in a 54-dimensional space [12]. We reduced the dimensionality to 7 via random projections [5], and took $200{,}000$ points in random order. **Mnist** is real data on handwritten digits with $8{,}100{,}000$ points in a 784 dimensional space [23]. We again reduced the dimensionality to 7, and took 200,000 points.

Nearest Neighbor (NN) is described in Section 3.2. This implementation saves a bounding box of all points within each node of the kd-tree, and splits nodes until there are four points or fewer in the leaf nodes. NN is an inferred order algorithm, but has *two* call sets. We use three inputs with separate training and test sets. For each point in the test set, we find the nearest neighbor in the training set. **Random** has a training set and test set of one million points each, randomly generated in a 7-dimensional space. We also used the **Covtype** and **Mnist** inputs described above, with a training set and test set of $200,000$ points each.

k-Nearest Neighbor (kNN) is an optimized k-nearest neighbors benchmark, using a different kd-tree variant than NN. This implementation does not save a bounding box per node. Instead it uses the difference between the split plane and the query point's corresponding dimension value as a termination condition. This condition is less aggressive in pruning nodes than the bounding box, but requires less computation. This implementation also saves the median point in non-leaf nodes, providing further speedup by reducing the number of nodes traversed. kNN is an inferred order algorithm with two call sets. We use the same three inputs as for NN, with $k = 5$.

Ball Tree (BT) is a nearest neighbor benchmark using a ball tree [31]. We use the implementation from the WEKA data mining software [18]. For efficiency reasons, our baseline uses an optimized distance computation that is tailored to our inputs. However, we did not alter the data structure or the actual traversal code; TreeSplicer was applied to the "out of the box" traversal algorithm. The distance computation kernel we modified is not touched by our transformations. BT is an inferred order algorithm with two call sets. We use the same three input sets as for NN, with **Random** having $600,000$ instead of one million points.

Ray Tracing (RT) can be accelerated with tree-structured bounding volume hierarchies (BVHs) that accelerate ray-object intersection tests. Our benchmark is extracted from the BVH-based ray tracer of Walter *et al.* [39]. Ray tracing is the most general benchmark we tackle, as it is *not* an inferred order algorithm and has *four* call sets, and hence we must track each call set explicitly[6]. However, it is pseudo-tail recursive. The input is a randomly generated scene with four million triangles. We rendered a single frame with 512×512 eye rays and 32 lights (hence, 32 shadow rays per eye ray).

[4] We included a benchmark, Lightcuts, in prior work to demonstrate the robustness of the transformation framework [19]. In that work, we were unable to run the benchmark with large enough inputs for locality transformations to be useful and so do not evaluate its performance in this work.

[5] CoV is the sample standard deviation divided by the mean.

[6] The non-inferred order with four call sets is due to the implementation we use. It is not a fundamental limitation of the BVH algorithm.

Benchmark	Lines of code	Transform time (ms)	Input	Traversal time %	Traversal time (s)	Heap usage			
						Base (MB)	SpliceElision (% increase)		
							-O0	-O1	-O2
Barnes-Hut	466	1045	Random	99.1	123.6	210	127.6	46.7	18.6
			Plummer	85.3	222.8	210	90.5	31.4	18.1
Point Correlation	396	1037	Random	99.6	895.2	188	132.4	21.3	2.1
			Covtype	99.0	776.3	59	96.6	18.6	8.5
			Mnist	94.4	147.4	56	107.1	21.4	5.4
Nearest Neighbor	450	1004	Random	98.7	380.1	390	47.2	2.6	
			Covtype	95.8	359.0	72	70.8	9.7	
			Mnist	96.6	476.8	74	60.8	2.7	
k-Nearest Neighbor	378	1114	Random	99.8	818.2	478	72.8	59.4	
			Covtype	85.1	74.2	92	78.3	52.2	
			Mnist	92.0	185.8	92	72.8	50.0	
Ball Tree	6199	2060	Random	99.1	1102.0	316	51.3	39.2	
			Covtype	93.5	235.8	84	90.5	76.2	
			Mnist	97.4	737.7	84	92.9	76.2	
Ray Tracing	3988	1960	Random	42.9	166.4	781	159.9		

Table 2. Transform time, traversal time and heap usage

6.1.2 Platforms

We evaluate our benchmarks on three systems with different cache configurations.

- The **Opteron** system runs Linux 2.6.24 and contains two dual-core AMD Opteron 2222 chips. Each chip has 128K L1 data cache per core and 1M L2 cache per core.
- The **Niagara** system runs SunOS 5.10 and contains two 8-core UltraSPARC T2 chips. Each chip has 8K L1 data cache per core and 4M shared L2 cache.
- The **Opteron II** system runs Linux 2.6.32 and contains four twelve-core AMD Opteron 6176 chips. Each chip has 64K L1 data cache per core, 512K L2 cache per core, and two 6M shared L3 caches.

6.1.3 Traversal times

For each of our benchmarks, we measure the amount of time spent in traversals, as this is the portion of code our transformations change. RT has two traversal phases, the first where initial rays are cast, and the second where shadow rays are traced. Because the initial rays in the first phase are sorted inherently, we measure the second phase to demonstrate the efficacy traversal splicing on unsorted points[7]. Column 5 of Table 2 shows the percentage of total time spent in traversals for each baseline benchmark/input on the Opteron II system. It can be seen that the traversal time is the dominant component of the total runtime. The exception in RT is because the BVH tree takes considerable time to build. We note that for real applications the same BVH will be used for multiple frames, making the traversal time dominant. Hereafter we will discuss the performance only of the traversal portions of the benchmarks.

6.1.4 Transformation times

Columns 2–3 of Table 2 shows the lines of code and transformation times for our benchmarks[8]. Our transformations are very quick, amounting to less than two seconds.

[7] The second phase accounts for 92.3% of the overall traversal time.

[8] Transformation times are on an Intel Core 2 Duo with blocking, splicing, and all optimizations applied.

6.2 Optimizations and heap usage

Traversal splicing requires additional space to save state of paused points. Columns 7–10 of Table 2 shows the heap usage of the baseline algorithms, and the percentage increase of the transformed, spliced implementations (with splice node elision) at three optimization levels:

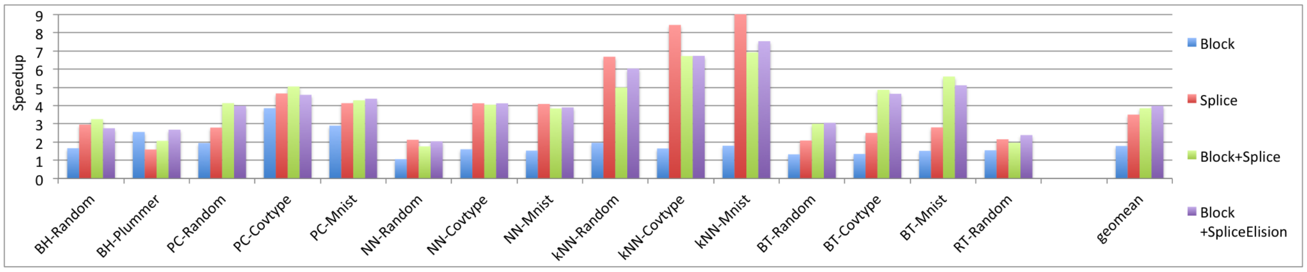
- **-O0** is the default spliced implementation for pseudo-tail recursive codes (Section 4.2).
- **-O1** is an optimized spliced implementation which exploits inferred order (Section 4.3).
- **-O2** is an optimized spliced implementation for codes with a single call set (Section A.3).

The percentage increase is left blank when the optimization level is not applicable to the algorithm. The memory overheads of splicing comes from extra state used to save four types of data: (i) paused points; (ii) call set ids for non-node-inferred-order algorithms; (iii) local variables in intermediary methods; and (iv) local variables within the recursive method. Type (i) is proportional to P (the number of points) and is always incurred for traversal splicing. **-O2** reduces the overhead of (i) by having a point buffer per level instead of a point buffer per node for **-O1**. (ii) is proportional to $P \times D$ and is incurred for **-O0**. (iii) is proportional to P and is incurred for BH, BT and RT. (iv) is proportional to $P \times D$ and is incurred for kNN. Hence we find that PC and NN at **-O2** which incur only (i) have the smallest heap increases, and RT which incurs (i), (ii) and (iii) has the largest heap increase. Note that the % increase depends on the size of points and local variables compared to the entire application footprint, so a direct comparison across benchmarks is difficult.

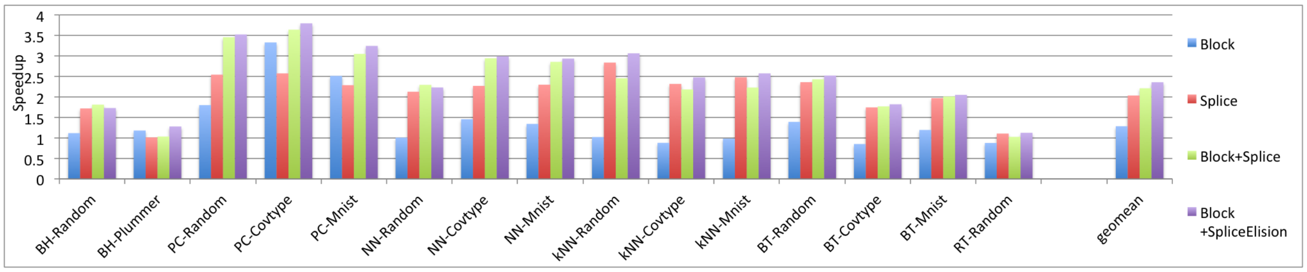
Not applying splice node elision increases the heap overhead by 10.4%, 29.5% and 2.8% respectively for **-O0**, **-O1** and **-O2**, over the heap size for splicing with splice node elision, as there are more phases and hence more state to save.

6.3 Serial results

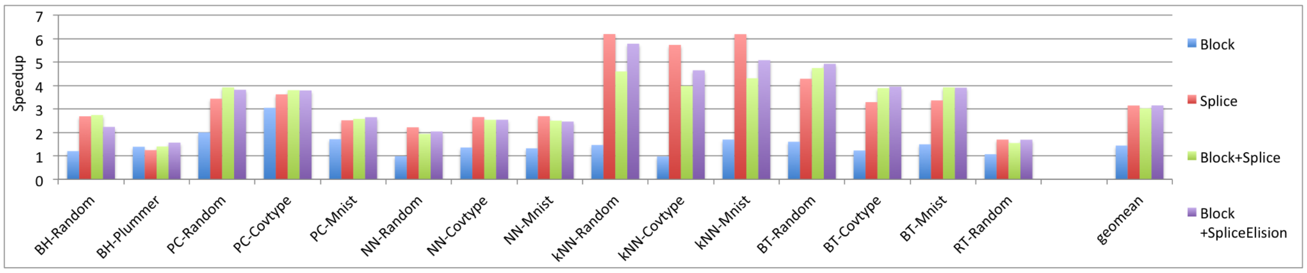
Figure 9 show speedups of **Block**, **Splice**, **Block+ Splice** and **Block+ SpliceElision** over **Base** for each benchmark/input pair, for the three systems. The geometric mean of the



(a) Opteron

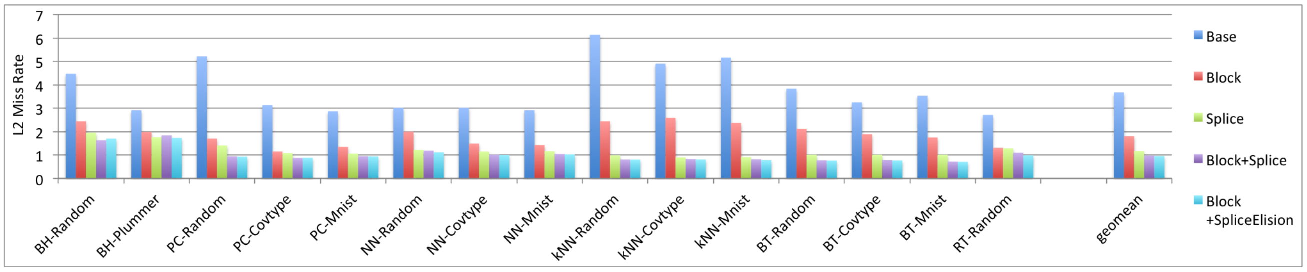


(b) Niagara

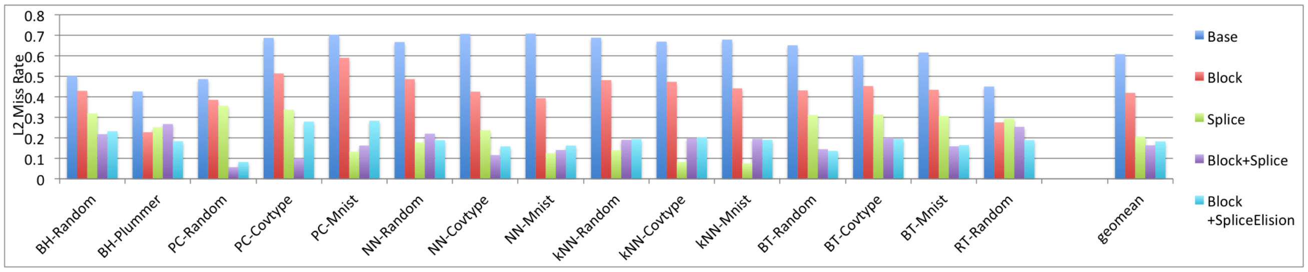


(c) Opteron II

Figure 9. Serial improvements: Speedup over **Base**



(a) CPI (Cycle per instruction)



(b) L2 miss rate

Figure 10. Performance counters

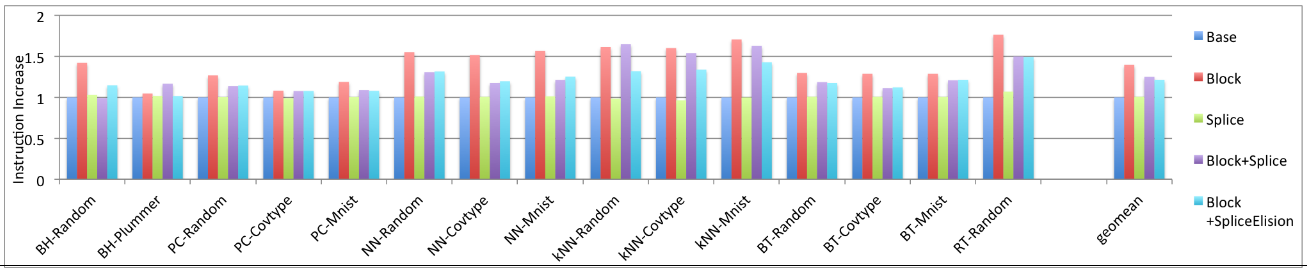


Figure 11. Performance counters: instruction counts normalized to **Base**

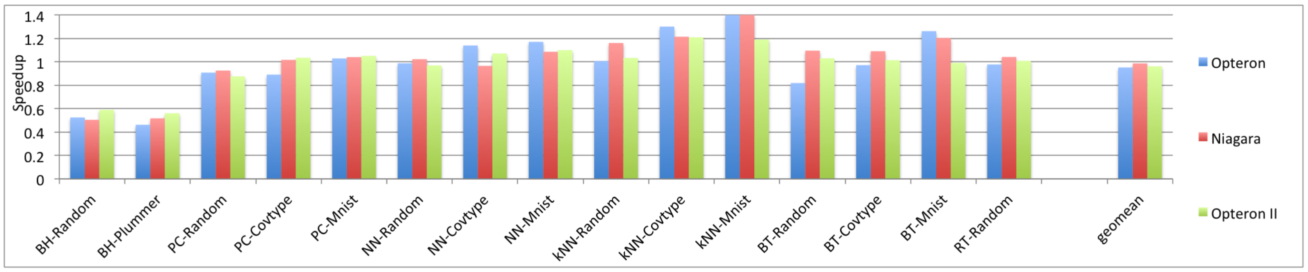


Figure 12. Speedup of **Block+Splice** over **Block+Sort**

speedups across all benchmark/input pairs on the Opteron are 1.787, 3.508, 3.857 and 3.992 for **Block**, **Splice**, **Block+ Splice** and **Block+ SpliceElision** respectively. The geometric mean of the speedups are 1.284, 2.034, 2.210 and 2.358 on the Niagara, and 1.440, 3.147, 3.022 and 3.151 on the Opteron II. Combined across all systems, the mean speedups are 1.489, 2.821, 2.953 and 3.095.

Figures 10 and 11 shows performance counter results on the Opteron II collected with PAPI [10]. The geometric mean of the L2 miss rate is 0.608 for **Base**. Applying point blocking in **Block** reduces this to 0.419. Applying traversal splicing in **Splice** independently brings the mean L2 miss rate to 0.206 and combining both point blocking and traversal splicing in **Block+Splice** reduces this to 0.163 which is less than a third of the baseline L2 miss rate. This enhanced locality is reflected in the CPI, which on average is 3.673 for **Base**, but is reduced to 1.813, 1.164 and 0.982 for **Block**, **Splice** and **Block+Splice** respectively. Further adding splice node elision increases the L2 miss rate slightly to 0.182 due to larger partial traversals, but decreases CPI to 0.961 likely due to L3 cache on the Opteron II.

The instruction increase of of **Block**, **Splice**, **Block+ Splice** and **Block+ SpliceElision** over **Base** is 1.396, 1.006, 1.249, 1.214 each. Point blocking often has substantial instruction overhead, because highly divergent points result in sparse blocks. The instruction overhead of applying splicing is almost negligible. Combining point blocking with traversal splicing reduces the instruction overhead significantly, as the dynamic sorting makes the blocks more dense. Adding splice node elision reduces the instruction overhead further.

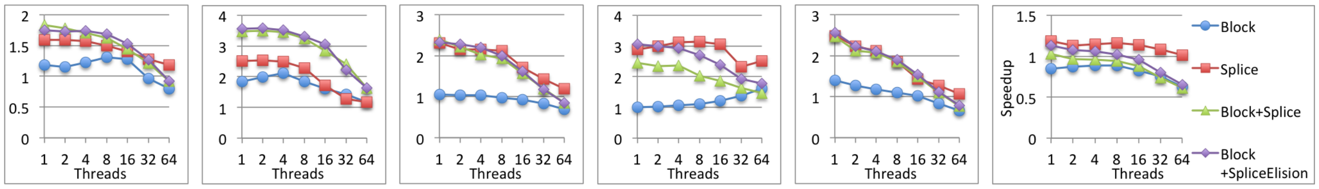
We see that point blocking is often an effective optimization, even without an *a priori* sorting pass. However, traversal splicing consistently outperforms blocking, often by significant amounts, due to its ability to reorder points on the fly. The improvements are particularly marked for kNN where we see speedups up to 9.147 for kNN-Mnist on the Opteron and 6.192 for kNN-Random on the Opteron II. On the Opteron II, this is because the L2 miss rate decreases 5-fold from 0.688 to 0.139, which lowers the CPI 6-fold from 6.13 to 0.99 for kNN-Random. In many cases the bottom phases still outstep cache even after traversal splicing. Further applying point blocking to the bottom phases can attain 13% more improvement over the baseline, and adding splice node elision squeezes out 14% more.

The inferred order optimizations described in Section 4.3 (**-O0**, **-O1** and **-O2** in Section 6.2) have insignificant impact on runtime, within error bounds of the confidence interval.

6.4 Comparison to manual optimization (sorting)

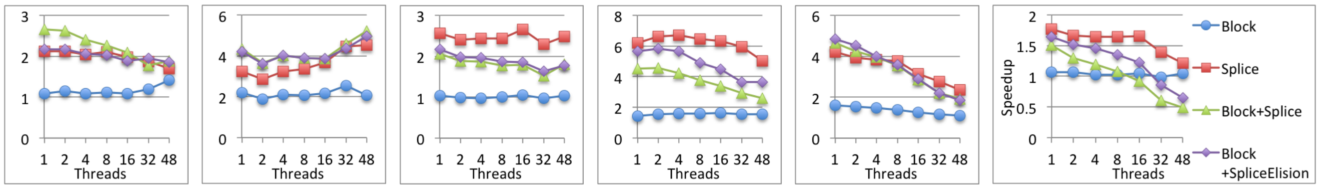
In the previous section, we showed that splicing can attain substantial improvements over both the baseline and point blocking with unsorted points. But how far are we from the performance of applications that have been hand-optimized, including with application-specific sorting? To explore this question, we compare the **Block+SpliceElision** version of each benchmark to a new version: **Block+Sort**, a hand-optimized implementation that applies point blocking (but not splicing) to *manually-sorted* points. In essence, this new version manually applies the best-available prior approaches for enhancing locality.

Sorting is done in tree order [36] for all benchmarks other than RT. For RT, the unsorted baseline uses the default ordering, where shadow rays are processed in the order they are produced (*i.e.*, in the order of the initial eye rays), while the sorted version groups shadow rays according to their light source. Sorting time is included for NN, kNN and BT where sorting has additional overhead because the points are different from the entities used to build the tree, amounting to between 0.3%–5.8% of the traversal time.



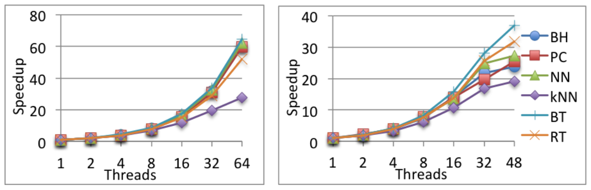
(a) Barnes-Hut (b) Point Correlation (c) Nearest Neighbor (d) kNN (e) Ball Tree (f) Ray Tracing

Figure 13. Speedup of transformed versions on n threads over **Base** on n threads on Niagara



(a) Barnes-Hut (b) Point Correlation (c) Nearest Neighbor (d) kNN (e) Ball Tree (f) Ray Tracing

Figure 14. Speedup of transformed versions on n threads over **Base** on n threads on Opteron II



(a) Scalability on Niagara (b) Scalability on Opteron II

Figure 15. Scalability of **Base**

Figure 12 shows the performance of **Block+SpliceElision** normalized to **Block+Sort**. The efficacy of manual, *a priori* sorting compared to traversal splicing varies with benchmark and system. Manual sorting is particularly effective for BH, whereas it is particularly ineffective for kNN. The geometric mean of the normalized runtimes is 0.951, 0.986 and 0.961 for the Opteron, Niagara and Opteron II systems respectively. We see that *across our 15 benchmark/input pairs, our fully automatic transformation is competitive with manual, application-specific optimization.*

While thus far we have pitched traversal splicing as an automatic optimization in the absence of semantic information, it remains effective even when combined with manual sorting. The speedups of the sorted counterparts, **Block+Sort**, **Splice+Sort**, **Block+Splice+Sort** and **Block+SpliceElision+Sort** over **Base+Sort** across all benchmarks and systems are 1.796, 1.650, 1.752 and 1.968 respectively. When the points are sorted, point blocking is generally better than traversal splicing, but a combination of blocking, splicing, and splice node elision can attain even more improvements. The overall improvements are less than for unsorted points, because unsorted points have worse locality to begin with, and hence more room for improvement. We present the full graphs for sorted improvements in Figure 16, in the Appendix.

6.5 Parallel improvements

TreeSplicer takes sequential code as input, and outputs sequential code[9]. To test our benchmarks on multicores, we manually parallelized each benchmark, with Java threads. **Base** and **Block** were parallelized by processing multiple points or blocks in parallel, and load balancing was applied with work stealing. **Splice**, **Block+ Splice** and **Block+ SpliceElision** are not amenable to load balancing as splicing requires having a large number of points up front, to exploit dynamic sorting among them. **Splice**, **Block+ Splice** and **Block+ SpliceElision** were parallelized by statically and uniformly distributing the points among threads, with each thread applying splicing to its portion of points. We also measured **Base** and **Block** without load balancing and found that load imbalance degrades performance for large number of threads. Hence the different parallelization strategy for splicing *detracts* from the improvements of splicing. We present results up to 64 threads for the Niagara, at which point we are employing 4-way multithreading, and up to 48 threads on the Opteron II. In the interests of brevity, we present results for the random inputs for all benchmarks.

We first present the scalability of the benchmark baselines in Figures 15(a) and 15(b), to show that the baselines do indeed scale. Figures 13 and 14 show parallel improvements of **Block**, **Splice**, **Block+ Splice** and **Block+ SpliceElision** over **Base** for each benchmark. While our techniques can deliver substantial improvements even in a parallel setting, the improvements generally decline as the number of threads increase. This is due to four factors. First, autotuning is currently done sequentially and limits speedup according to Amdahl's law. While we have limited the autotuning overhead to 1% for the serial case, parallelization can increase its overhead to up to 30% for 64 threads. Second, splic-

[9] TreeSplicer could automatically transform parallel code and apply the appropriate parallelization strategy, if a particular parallelization scheme were integrated into it. We currently do not implement this.

ing's dependence on exploring large numbers of points to exploit reordering means that its relative improvement drops as scale increases, as each thread processes fewer points. This effect should be mitigated for larger inputs, where each thread will receive more points. Third, the static distribution of work leads to significant load imbalance at large numbers of threads. Finally, hardware multithreading hides miss latency and attenuates the importance of locality optimizations. The first issue affects all autotuned versions, while the second and third affect only the spliced versions. The last issue affects all versions on the Niagara at 32–64 threads.

7. Related Work

Much of the work on optimizing locality in irregular algorithms has focused on *scheduling* computation so that tasks likely to access similar data are scheduled in close succession to exploit temporal locality. This has been the strategy of choice for optimizing sparse-matrix algorithms [9, 27, 37], where most approaches use an *inspector-executor* approach to scheduling. The structure of the computational tasks is found in an inspection phase, which rearranges them to improve locality. The rearranged schedule is then executed. Inspector-executor approaches are less useful for tree traversal codes, as the inspection phase requires performing the traversals, incurring all the misses we hope to avoid.

Scheduling approaches for tree traversals (the "sorting" optimizations we discuss in Section 2.1) have instead used semantic knowledge to schedule the points without performing the traversals, often with space filling curves [2, 28]. Singh *et al.* order the points in Barnes-Hut according to their position in the oct-tree [36].Mansson *et al.* propose various sorting optimizations for reflected rays in ray tracing [24]. In fact sorting for ray tracing is difficult and important enough that Moon *et al.* first construct and traverse a *simplified* scene, and sort rays based on hit points of the simplified scene [28].

There have been a few application-specific, manual approaches similar to traversal splicing, targeting locality for ray tracing. Pharr *et al.* partition the scene into a uniform grid of "voxels" [32]. As rays traverse a scene, they pause at the boundaries of voxels and are resumed later. By processing rays on a per-voxel basis, locality is improved. They schedule voxels based on a cost-benefit heuristic which considers the amount of geometry already cached and the number of the rays paused at the voxel. Navrátil *et al.* pause traversals at "queue points" (akin to splice nodes) in the tree [30]. Queue points are placed so that its subtree fits entirely in L2 cache. Navrátil's thesis focuses on dynamic scheduling of rays to enhance locality for memory efficiency and scalability across thousands of distributed processors [29]. Aila and Karras extend Navrátil *et al.*'s work to hierarchical queue points and massive parallelism [1]. They use dynamic programming to place treelets (subtrees at which rays are paused) with balanced surface area, hence minimizing treelet transitions per ray, and discuss considerations for a dedicated GPU architecture which supports efficient pausing and resuming of rays. These approaches are able to make smarter decisions on splice node placement and point scheduling (*e.g.,* next voxel to process) with semantic knowledge of the algorithm. We believe TreeSplicer could potentially make such smart decisions automatically without semantic knowledge, through a dynamic analysis of traversal patterns at runtime.

Pingali *et al.* propose *computation reordering*, whereby an individual computation can be paused during its execution and coalesced with other computations that are accessing the same part of the data structure [33]. However, computation reordering is more a set of principles for optimization than an optimization itself: correctly applying reordering requires manually transforming algorithms in application-specific ways. Traversal splicing can be seen as a special, disciplined case of computation reordering, applying to tree traversals, that can be implemented automatically and efficiently.

Most prior compiler efforts targeting irregular programs have focused either on analysis, like shape analysis [15, 35], or parallelization [14, 26]. Rinard and Diniz prove that the traversal loop is parallelizable through commutativity analysis, whereas traversal splicing requires that the point loop be parallelizable [34]. These approaches are complementary to ours. Our automatic transformation framework can benefit from analyses to identify tree-shaped data structures or parallelizable loops over irregular data structures, allowing us to infer properties we currently identify through annotation.

A large number of prior studies have investigated improving *spatial* locality in irregular algorithms. Truong *et al.* propose *ialloc* to facilitate programmer driven field reorganization and instance interleaving, where frequently used "hot" fields of many instances are interleaved, so that "cold" fields do not occupy the same cache line as hot fields [38]. Chillimbi *et al.* automate this process for Java programs with a profiling and program transformation tool, and automatically generate field reordering recommendations for C programs [6]. Separate work by Chillimbi *et al.* propose *ccmorph* to reorganize memory layout of trees and *ccmalloc* to collocate objects based on programmer provided hints [7], and perform relayout during garbage collection [8]. Lattner and Adve use a context-sensitive pointer analysis to segregate distinct instances of heap allocations into separate memory pools, and thereby enhance locality at the *macroscopic* level of entire data structures [22]. Wang *et al.* also aggregate affinitive heap objects into dedicated memory regions, but do so dynamically without access to the source code [40]. Because these approaches focus on data layout, they do not target temporal locality, as the transformations presented in this paper do. We expect our transformations to be complementary to these spatial-locality-enhancing approaches.

8. Conclusions

We presented traversal splicing, a comprehensive, general, automatic transformation that applies to tree traversal codes such as Barnes-Hut and nearest neighbor. Unlike previously proposed transformations, such as point sorting or point blocking, the effectiveness of traversal splicing is not dependent on first performing any application-specific, semantics-aware hand transformation. Furthermore, traversal splicing exploits the insight that during a point's traversal of a tree, its remaining traversal structure can be predicted from its past behavior. Hence, we can splice together traversals from many points, using this predictive property to group points with similar traversals together.

We showed that when presented with baseline algorithms, our automated traversal splicing transformations outperformed, often substantially, previously presented transformations for traversal codes. In fact, we showed that applying traversal splicing to traversal algorithms, even in the absence of any hand optimization, can yield results competitive with manually-transformed, locality-enhanced implementations.

We note that while we only apply traversal splicing to tree traversal algorithms, there is nothing, in principle, preventing a similar optimization from being applied to any irregular traversal algorithm. Splice nodes can be viewed as "boundary" nodes in a tree, and traversals that reach boundary nodes are paused until a later date. One could place similar boundary nodes in a graph by performing graph partitioning. A graph traversal (*e.g.* a graph search) would then operate within a single partition, and, when it reached a partition boundary, would be paused, only to be resumed later when other traversals accessing the same partition were found. While the scheduling complexity of applying a splicing-like algorithm to general graphs is substantially higher than for trees, there is nevertheless a possibility of deploying an even more general version of traversal splicing. This is a promising area for future work.

Acknowledgments

This work was supported in part by an NSF CAREER Award (CCF-1150013) and grants from the Purdue Research Foundation and Intel. The authors would like to thank Michael Goldfarb, Hyokun Yun and the anonymous referees for their useful comments. We are grateful to Bruce Walter for providing the Ray Tracing benchmark code, and Vijay Pai and his students for providing support for machines on which our tests were conducted.

References

[1] T. Aila and T. Karras. Architecture considerations for tracing incoherent rays. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 113–122, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[2] M. Amor, F. Argüello, J. López, O. G. Plata, and E. L. Zapata. A data parallel formulation of the barnes-hut method for n -body simulations. In *Proceedings of the 5th International Workshop on Applied Parallel Computing, New Paradigms for HPC in Industry and Academia*, pages 342–349, 2001.

[3] J. Barnes and P. Hut. A hierarchical $o(nlogn)$ force-calculation algorithm. *Nature*, 324(4):446–449, December 1986.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509–517, September 1975.

[5] E. Bingham and H. Mannila. Random projection in dimensionality reduction: applications to image and text data. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '01, pages 245–250, New York, NY, USA, 2001. ACM.

[6] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 13–24, 1999.

[7] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 1–12, 1999.

[8] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the 1st international symposium on Memory management*, pages 37–48, 1998.

[9] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 229–241, 1999.

[10] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using papi for hardware performance monitoring on linux systems. In *In Conference on Linux Clusters: The HPC Revolution, Linux Clusters Institute*, 2001.

[11] T. Ekman and G. Hedin. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 1–18, 2007.

[12] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[13] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[14] R. Ghiya, L. Hendren, and Y. Zhu. Detecting parallelism in c programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1:35–47, 1998.

[15] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15, 1996.

[16] A. G. Gray and A. W. Moore. N-Body Problems in Statistical Learning. In T. K. Leen, T. G. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems (NIPS)*

13 (Dec 2000), 2001.

[17] M. Greenspan and M. Yurick. Approximate kd-tree search for efficient ICP. In *Fourth International Conference on 3-D Digital Imaging and Modeling*, pages 442–448, 2003.

[18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

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

[20] K. Kennedy and J. Allen, editors. *Optimizing compilers for modren architectures:a dependence-based approach*. 2001.

[21] M. Kulkarni, M. Burtscher, K. Pingali, and C. Cascaval. Lonestar: A suite of parallel irregular programs. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 65–76, April 2009.

[22] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 129–142, 2005.

[23] G. Loosli, S. Canu, and L. Bottou. Training invariant support vector machines using selective sampling, 2005.

[24] E. Mansson, J. Munkberg, and T. Akenine-Moller. Deep coherent ray tracing. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 79–85, 2007.

[25] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[26] L. A. Meyerovich, T. Mytkowicz, and W. Schulte. Data parallel programming for irregular tree computations. In *3rd USENIX workshop on hot topics in parallelism*, 2011.

[27] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 192–, 1999.

[28] B. Moon, Y. Byun, T.-J. Kim, P. Claudio, H.-S. Kim, Y.-J. Ban, S. W. Nam, and S.-E. Yoon. Cache-oblivious ray reordering. *ACM Trans. Graph.*, 29(3):28:1–28:10, July 2010.

[29] P. A. Navratil. *Memory-efficient, scalable ray tracing*. PhD thesis, 2010.

[30] P. A. Navratil, D. S. Fussell, C. Lin, and W. R. Mark. Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pages 95–104, Washington, DC, USA, 2007. IEEE Computer Society.

[31] S. M. Omohundro. Five balltree construction algorithms. Technical report, 1989.

[32] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 101–108, 1997.

[33] V. K. Pingali, S. A. McKee, W. C. Hseih, and J. B. Carter. Computation regrouping: restructuring programs for temporal data cache locality. In *Proceedings of the 16th international conference on Supercomputing*, pages 252–261, 2002.

[34] M. Rinard and P. C. Diniz. Commutativity analysis: a new analysis technique for parallelizing compilers. *ACM Trans. Program. Lang. Syst.*, 19(6):942–991, 1997.

[35] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3), May 2002.

[36] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. *J. Parallel Distrib. Comput.*, 27(2):118–141, 1995.

[37] M. M. Strout, L. Carter, and J. Ferrante. Rescheduling for locality in sparse matrix computations. In *Proceedings of the International Conference on Computational Sciences-Part I*, pages 137–148, 2001.

[38] D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 322–, 1998.

[39] B. Walter, K. Bala, M. Kulkarni, and K. Pingali. Fast agglomerative clustering for rendering. In *IEEE Symposium on Interactive Ray Tracing (RT)*, pages 81–86, August 2008.

[40] Z. Wang, C. Wu, and P.-C. Yew. On improving heap memory layout by dynamic pool allocation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 92–100, New York, NY, USA, 2010. ACM.

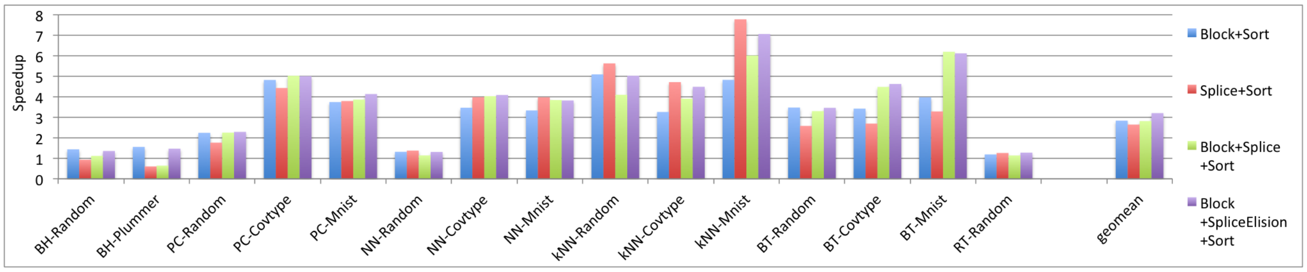
A. Detailed splicing implementation

This section presents a more in-depth description of the implementation of traversal splicing. We use the nearest neighbor (NN) algorithm of Figure 5 as a running example, and the combinations of Figures 17, 18 and the `recurse` method of Figure 5 comprises the full spliced implementation.

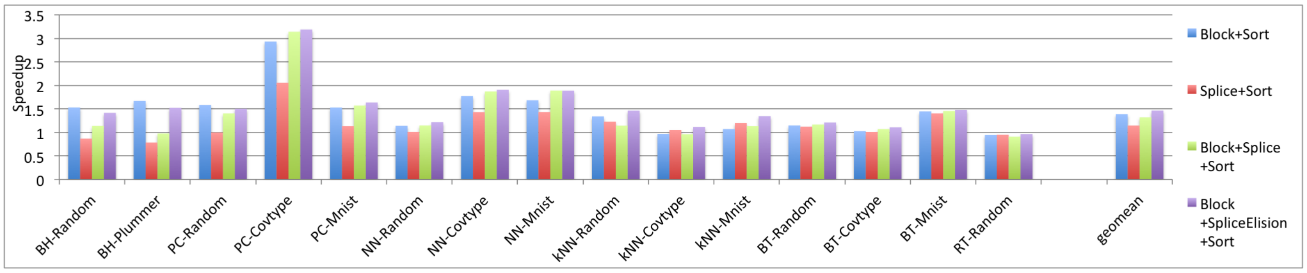
A.1 Setup and recursion unrolling

In the actual implementation, the high level view of Figure 7 is moved to lines 1-8 of Figure 17. We map the top of the tree and save the nodes into an array based on a heap ordering of the tree as in Figure 2(a). Non-existent nodes are marked null in the array. Point buffers into which points can be saved are allocated for all non-null nodes. Both the tree map and point buffers are indexed from 1 (the root) to $C^{D+2} - 1$ (the rightmost leaf), where C is `maxChildren`, and D is the splice depth. Point buffers need to accommodate an arbitrary number of points and are implemented as ArrayLists. The call set id buffer is a two dimensional array of size $P \times D$ (P is the number of points).

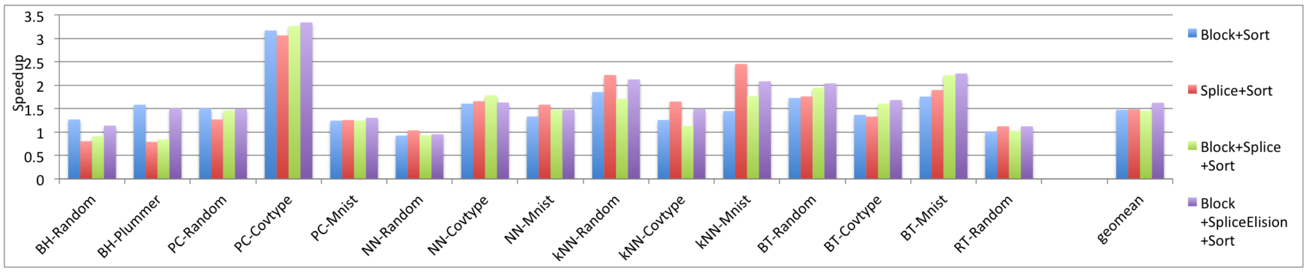
The partial traversals of all points are executed by pairs of top and bottom phases. While bottom phases are equivalent to normal recursive traversals, top phases execute a path through the tree down to the splice depth, and needs to save additional state so that traversals can be resumed. Figure 18 shows a *transformed* recursive method, `recurseSplice`, to realize top phases for the pseudocode for the nearest neighbor code of Figure 5. `recurseSplice` performs



(a) Opteron



(b) Opteron



(c) Opteron II

Figure 16. Serial improvements: Speedup over **Base+Sort**

only the *first* traversal of each call set, later traversals in the call sets will be directly called (on the appropriate child) in subsequent phases. Points that reach the splice depth are saved into the node at which they should be *resumed* (not the node at which the point is paused) (lines 14 and 21), and points that are truncated beforehand are saved into the node at which they are truncated (lines 3 and 7). The call set id is saved per point per depth (lines 16 and 23).

Because the transformed code performs only the *first* traversal of a call set, subsequent partial traversals of the tree above the splice nodes are performed by later top phases. Top phase traversals are paused at explicit splice nodes, and the traversal is resumed by bottom phases that continue the traversal to the implicit splice node. The order of phases can be determined by partially unrolling the recursive traversal (`unrollRecursion` in lines 19-29 in Figure 17), which basically expands the remaining top phases and appends a bottom phase after each top phase. The first top phase is started by calling `recurseSplice` for all points (line 6 of Figure 17. The depth of the nodes at which to resume and the phase number are passed as arguments to the top phase. For a bottom phase, the depth argument will always be $D + 1$, and all phase numbers are iterated upon. For our running example (Figures 4 and 6(a), $D = 2$, root is depth 0), the order of phases is T0-0 (top phase at depth 0, traversal phase 0), B (bottom phase), T2-1, B, T1-1, B, T2-1, B (as in Figure 6(a)). The first top phase always performs traversal phase 0 (the first traversal), and because our example has two phases per call set, all other top phases start with the second traversal in the set.

A.2 Dynamic sorting

In our implementation the dynamic sorting comes naturally, as points are reordered during top phases based on the node they were saved at. At each top phase, we gather all points which should execute in this phase (*i.e.*, all points that have not been truncated *above* the level of the top phase), and resume them at the appropriate node based on the point's call set id (lines 37-50 of Figure 17). For example at T2-1, we will gather points that have been paused or truncated at nodes ④–⑮, and resume at nodes ④–⑦. At T1-1 we will gather points at nodes ②–⑮, and resume at nodes ② and ③.

More specifically for our particular example of NN, at T2-1, we will gather points at nodes ④, ⑧, ⑨ and resume at node ⑤, because these points have visited {leftChild}

```
 1 Set<Point> points = /* points */
 2 Node root = buildTree(points);
 3 mapTree(root);
 4 allocBuffers();
 5 foreach (Point p : points) {
 6   recurseSplice(p, root, 0);
 7 }
 8 unrollRecursion(0, 0, 0, 0);
 9
10 void allocBuffers() {
11   numLeafNodes = power(maxChildren, D + 1);
12   numNodes = numLeafNodes * maxChildren;
13   for (int i = 1; i < numNodes; i++) {
14     if (treeMap[i] != null) allocPointBuffers(i);
15   }
16   allocCallSetIdBuffer();
17 }
18
19 void unrollRecursion(int d1, int p1, int d2, int p2) {
20   if (d2 < D) {
21     for (int i = 0; i < maxPhases; i++) {
22       if (i == 0) unrollRecursion(d1, p1, d2 + 1, i);
23       else unrollRecursion(d2 + 1, i, d2 + 1, i);
24     }
25   } else {
26     if (d1 != 0) topPhase(d1, p1);
27     bottomPhase();
28   }
29 }
30
31 void topPhase(int depth, int phase) {
32   int start = power(maxChildren, depth);
33   int end = start * maxChildren;
34   for (int i = start; i < numNodes; i++) {
35     swapPointBuffersAndClearDest(i);
36   }
37   for (int i = start; i < end; i++) {
38     int substart = i;
39     int subend = i + 1;
40     while (substart < numNodes) {
41       for (int j = substart; j < subend; j++) {
42         foreach(Point p : getPointsAtNodeIndex(j)) {
43           Node n = nextNode(p.getCallSet(depth), phase);
44           recurseSplice(p, n, depth);
45         }
46       }
47       substart *= maxChildren;
48       subend *= maxChildren;
49     }
50   }
51 }
52
53 void bottomPhase() {
54   for (int i = numLeafNodes; i < numNodes; i++) {
55     foreach(Point p : getPointsAtNodeIndex(i)) {
56       for (int j = 0; j < maxPhases; j++) {
57         Node n = nextNode(p.getCallSet(D), j);
58         recurse(p, n);
59       }
60     }
61   }
62 }
```

Figure 17. Pseudocode of recursion unrolling

and should now visit {rightChild}. The points which resume at ⑤ are better sorted now because they are in order of truncation at ④, reach ⑧ first, and reach ⑨ first. Similarly, points at nodes ⑤, ⑩, ⑪ resume at node ④, points at nodes ⑥, ⑫, ⑬ resume at node ⑦, and so on.

To separate the points saved in the previous phase which need to be processed, and the points being saved in this phase, we use a toggle mechanism with *two* point buffers per node. At the start of a top phase, all point buffers below the depth argument are swapped, so the previous destination

```
void recurseSplice(Point p, Node n, int depth) {
  if (!canBeCloser(p, n.boundingBox)) {
    n.savePoint(p);
    return;
  } else if (n.isLeaf()) {
    p.updateClosest(n.getPoint());
    n.savePoint(p);
  } else {
    double split = p.value(n.splitType);
    if (split <= n.splitValue) {
      if (depth < D) {
        recurseSplice(p, n.leftChild, depth + 1);
      } else {
        n.leftChild.savePoint(p);
      }
      p.saveCallSet(depth, 0); // call set 0
    } else {
      if (depth < D) {
        recurseSplice(p, n.rightChild, depth + 1);
      } else {
        n.rightChild.savePoint(p);
      }
      p.saveCallSet(depth, 1); // call set 1
    }
  }
}
```

Figure 18. Transformed recursive method for top phases

buffer is the source buffer and vice versa (lines 34-36). Then the new destination buffer is cleared, and points will be saved to it during this phase. This is only done for top phases, as bottom phases do not save state.

For each bottom phase, all points that were paused at the explicit splice nodes (and saved into the children of explicit splice nodes, nodes ⑧–⑮) will traverse the appropriate subtrees below the splice nodes according to their call set id (lines 54-61). There is no sorting at bottom phases as points are gathered from a single node.

A.3 Optimizations

For inferred order algorithms, as in nearest neighbor, the call set id is unnecessary, and the next node can be determined completely by the previous node the point was at, and the phase number. Hence `i` can replace `p.getCallSet()` (lines 43 and 57 in Figure 17), and loop invariant statements can be hoisted out of the loop.

For single call set algorithms, recursion unrolling can be simplified even further. We need not scan through multiple subtrees as only a single path from the root will have saved points at a time. Hence `unrollRecursion` can pass a `nodeIndex` to both the top and bottom phases to indicate where to look for points. Furthermore, we need only one point buffer per level, instead of one point buffer per node.

Splice node elision can be realized by executing a *merged* bottom phase at line 23 of Figure 17, if the depth is a fixed distance from D. The merged bottom phase uses the code for `topPhase` but doesn't swap point buffers and calls `recurse` instead of `recurseSplice`.

B. Automatic transformation

Automatically transforming traversal codes to apply splicing first requires recognizing the traversal structure. While

Benchmark	Input	Block Sizes						Splice Depths					
		Opteron		Niagara		Opteron II		Opteron		Niagara		Opteron II	
		Block	Block +Splice	Block	Block +Splice	Block	Block +Splice	Splice	Block +Splice	Splice	Block +Splice	Splice	Block +Splice
BH	Random	512	512	512	512	512	512	2	2	2	2	2	2
	Plummer	512	512	512	512	512	512	3	3	3	3	3	3
PC	Random	512	512	512	512	512	512	8	8	8	8	8	8.6
	Covtype	128	128	128	128	128	128	7	7	7	7	7	7
	Mnist	128	128	128	128	128	128	7	7	7	7	7	7
NN	Random	512	512	512	512	512	512	7	7	7	7	7	7
	Covtype	128	128	128	128	128	128	7	7	7	7	7	7
	Mnist	128	128	128	128	128	128	7	7	7	7	7	7
kNN	Random	512	512	170.67	512	512	512	8	8	8	8	8	8
	Covtype	89.6	128	0	128	51.2	128	7	7	7	7	7	7
	Mnist	128	128	42.67	128	128	128	7	7	7	7	7	7
BT	Random	512	512	512	512	512	512	6	6	6	6	6	6
	Covtype	128	128	128	128	128	128	6	6	6	6	6	6
	Mnist	128	128	128	128	128	128	6	6	6	6	6	6
RT	Random	1024	1024	1024	1024	1024	1024	8	8	8	8	8	8

Table 3. Autotuned parameters

recognizing a traversal code structure can be expedited with programmer annotations, many traversal codes have a common algorithmic structure that does not require annotations to recognize. In particular, many traversal codes are written by recursive function calls on recursive data structures. TreeSplicer identifies code which performs *repeated recursive traversals of a recursive structure* as a candidate for traversal splicing. A recursive method, m, can be recognized by finding a call to itself within a method's body[10]. A recursive structure can be recognized by finding a class, c, with at least one field f of the same class (or superclass).

TreeSplicer then determines whether the recursive method performs a recursive traversal of any identified recursive structures. This might happen in one of two ways: (i) if m takes an object o of class c as an argument, and passes $o.f$ as an argument to the recursive call; or (ii) if m is a member method of c and it performs the recursive call by invoking $f.m()$ (in other words, the data structure node is the implicit "this" argument). TreeSplicer uses a call graph analysis to determine that the recursive method is called (either directly, or through a chain of calls) from a loop in the application. If the loop is annotated as parallel and the recursive structure is annotated as a tree, TreeSplicer transforms the code as described in Sections 5.3 and A.

Figure 17 is a template customized based on the optimization levels which is plugged in to the transformed code. Figure 18 is constructed by making a duplicate of the original recursive method, stripping all calls other than the *first* call of each call set, and adding code to save points and call set ids. Intermediary methods are split around the call path to the recursive method, into prologues and epilogues. Code is added to save any local state which persists from prologue to epilogue, and local variables within the recursive method which is passed as an argument to recursive calls. Autotuning code is added before any splicing setup is done (before line 3 of Figure 17), as the setup requires the splice depth, which is determined by the autotuner. Points consumed for autotuning are skipped in the point loop (lines 5-7).

We have made the source code of TreeSplicer public at https://sites.google.com/site/treesplicer.

C. Autotuned parameters

Table 3 shows the autotuned parameters averaged across the recorded serial runs on the three systems. The maximum block size is limited to 0.1% of the total points, so that each block size can be tested 5 times, and the autotuning overhead limited to 1% of the points. The selected block sizes are often capped at the maximum (512 for random inputs with one million points, and 128 for real inputs with $200,000$ points, 1024 for RT with 2^{23} points), because the points are unsorted, and the blocks become sparse quickly.For kNN the divergence is worse enough that sometimes the autotuner decides not to do point blocking. The splice depths are determined by the average reach of points, and is machine independent. Slight differences are due to the random sampling of points for autotuning. BH has a smaller depth because it is an octtree. For other benchmarks, random inputs have larger depths than real inputs, because there are more points, and the tree is larger.

[10] A more sophisticated approach is to look for cycles in a call graph; the simple approach here suffices for our benchmarks.