2-15-2012

Automatically Enhancing Locality for Tree Traversals with Traversal Splicing

Youngjoon Jo
Electrical and Computer Engineering, Purdue University, yjo@purdue.edu

Milind Kulkarni
Purdue University - Main Campus, milind@purdue.edu

Automatically Enhancing Locality for Tree Traversals with Traversal Splicing

Youngjoon Jo

Milind Kulkarni

TR-ECE-12-03

February 15, 2012

School of Electrical and Computer Engineering

1285 Electrical Engineering Building

Purdue University

West Lafayette, IN 47907-1285

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 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, recent work has proposed *point blocking*, a technique analogous to loop tiling in regular algorithms, to improve locality. However, point blocking requires that programs be "pre-optimized" using application-specific techniques to be effective. In this work, we identify the root cause of point blocking's poor performance on baseline irregular algorithms, and propose *traversal splicing*, a new, general, automatic locality optimization for irregular tree traversal codes, that addresses these drawbacks. For four benchmark algorithms, we show that traversal splicing can deliver substantial single-thread performance improvements of up to 338% (geometric mean: 138%) over baseline implementations, and up to 112% (geometric mean: 77%) over point-blocked implementations. Further, we show that in many cases, applying traversal splicing to a baseline implementation yields performance that is competitive with carefully hand-optimized implementations.

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 [15], 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. Most work in this area has focused on *ad hoc* techniques that leverage application semantics [1, 18, 21, 22, 25], or work well for sparse-matrix style algorithms [7, 20, 26], but not the tree- and graph-based algorithms that proliferate in domains like data mining and graphics. What we would like are *general transformation techniques to improve 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 [2], data mining algorithms such as point correlation and nearest neighbor [12], and graphics algorithms like ray tracing [28]. 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.

Recent work by Jo and Kulkarni discussed an abstract model of tree traversal codes that analogizes them to doubly-nested loops as seen in regular algorithms like vector outer product [14]. 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, Jo and Kulkarni propose 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.

A major drawback to point blocking, however, is that it is not truly automatic, nor is it application-agnostic. The transformation is applied on top of algorithms that implement a *point sorting* optimization, whereby points that have similar traversals are scheduled in close succession [1, 25]. 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.

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. Hence, applying point blocking, which relies on point sorting, is unlikely to be effective in a majority of cases. 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. A key feature of traversal splicing is that the order in which partial traversals are performed can be changed during execution. In particular, as points traverse the tree, we use their traversal history as a predictor of their remaining traversal patterns. Thus, *as points traverse the tree* we can 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 the 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 four benchmark algorithms, and show that traversal splicing delivers (single-thread) performance improvements of up to 338% (geometric mean: 138%) when compared to straightforward implementations of these algorithms, and up to 112% (geometric mean: 77%) 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 in many situations, *applying traversal splicing to a naïve implementation of an algorithm gives comparable or better performance than a hand-optimized implementation*.

Contributions

The contributions of this paper are:

- The development of *traversal splicing*, a new, general transformation for tree traversal codes that improves on prior work by effectively transforming 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, in some cases, 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 [2], nearest neighbor [12], iterative closest point [13] and many ray tracing algorithms [28], 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 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 [12].

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* [3]. 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.

```
1 Set<Point> points = /* points */
2 KDNode root = buildTree(points);
3 foreach (Point p : points) {
4   recurse(p, root);
5 }
6
7 void recurse(Point p, KDNode n) {
8   if (!canCorrelate(p, n.boundingBox)) {
9     return
10  } else if (n.isLeaf()) {
11    p.updateCorrelation(n.getPoint());
12  } else {
13    recurse(p, n.leftChild);
14    recurse(p, n.rightChild);
15  }
16 }
```

Figure 1. Pseudocode of point correlation

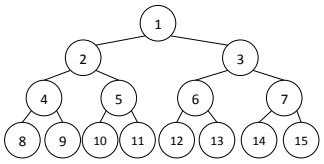
```
1 Set<Point> points = /* points */
2 foreach (Point p : points) {
3   foreach (KDNode n : p.oracleNodes()) {
4     if (n.isLeaf()) {
5       p.updateCorrelation(n.getPoint());
6     }
7   }
8 }
```

Figure 2. Point correlation as doubly nested loop

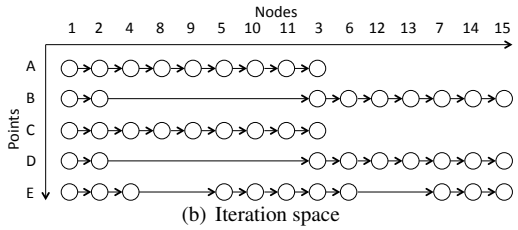
While all tree traversal algorithms we consider have the same basic structure, they each traverse their trees according to different criteria and use trees with different structures (oct-trees for Barnes-Hut, kd-trees for nearest neighbor, bounding volume hierarchies for ray tracing), and dynamically 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, Jo and Kulkarni suggest eliding the traversal pattern, and indeed the tree structure itself, and instead considering each point's traversal as though its path was provided by some oracle [14], thus viewing each algorithm as a simple doubly-nested loop, as shown in Figure 2. 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 3(a) shows a sample kd-tree for PC, with nodes numbered in heap order, and figure 3(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 [19] 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.

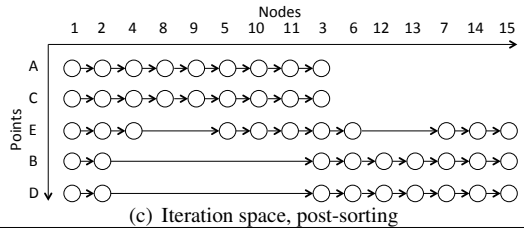
One popular approach to improving the locality of tree-traversal codes is to "sort" points so that points with similar traversals are executed close together, as in Figure 3(c), which shows the same traversals as Figure 3(b), but in a different order. Because points have different traversals, this sorting 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



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



(b) Iteration space



(c) Iteration space, post-sorting

Figure 3. Sample tree and iteration spaces

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 [25] or space-filling curves [1]. 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.

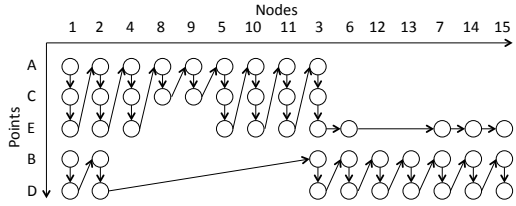
2.2 Point blocking

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. *Point blocking* was introduced by Jo and Kulkarni as a method for improving locality for tree traversal codes when traversal sizes attenuate the benefits of sorting [14]. The essence of point blocking is to "tile" the point loop, yielding the pseudocode shown in Figure 4(a). Compared to the base algorithm of Figure 1, the code is similar except the recursive method operates over a block of points rather than a single point. Rather than finishing an entire traversal for one point before moving on to the next point, a block of points moves through the tree in lockstep. The block visits nodes in the tree comprising the union of its component points' traversals, and points within the block only interact with nodes they would have in the original code. If none of the points in a block interact with a node, the block will skip visiting the node. The arrows in Figure 4(b) show the new iteration order when applying point blocking to the sorted traversals of Figure 3(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

```
1 Set<Point> points = /* points */
2 KDNode root = buildTree(points);
3 foreach (Block<Point> b : points) {
4   recurse(b, root);
5 }
6
7 void recurse(Block b, KDNode n) {
8   Block<Point> nextB; //points that continue traversing
9   for (int i = 0; i < b.size; i++) {
10    Point p = b.p[i];
11    if (!canCorrelate(p, n.boundingBox)) {
12      continue;
13    } else if (n.isLeaf()) {
14      p.updateCorrelation(n.getPoint());
15    } else {
16      nextB.add(p);
17    }
18  }
19  if (nextB.size > 0) {
20    recurse(nextB, n.leftChild);
21    recurse(nextB, n.rightChild);
22  }
23 }
```

(a) Pseudocode for point blocking



(b) Iteration order for point-blocked code

Figure 4. Point blocking

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 4(b)), 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 3(b), we note that two blocks would have to visit ④, resulting in two misses.

For point blocking to perform well, it needs a high *effective block size*. Effective block size is the average number of points per block that interact with each node of a traversal. A high effective block size relative to the actual block size indicates that the points in the block have similar traversals, whereas a low effective block size indicates that the points have highly divergent traversals. Table 1 shows the *normalized effective block size* (ratio of effective block sizes to actual block sizes) for point blocking with and without sorting applied on four benchmarks—Barnes-Hut, point correlation, nearest neighbor, and ray tracing. The table also shows the 1-thread performance of point blocking on both sorted and unsorted versions of the benchmarks run on the Opteron system of Section 6.

While point blocking can be effective on unsorted inputs, we see that combining point blocking and sorting dramatically improves effective block size and 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 that can be used to improve the effective block size of point-blocked code.

Benchmark	Normalized Effective Block Size		Runtimes (seconds)		
	Unsorted	Sorted	Unsorted Baseline	Unsorted Blocking	Sorted Blocking
Barnes-Hut	0.0044	0.1194	127.6	35.6	25.4
Point Correlation	0.0067	0.3059	784.9	396.2	179.3
Nearest Neighbor	0.0022	0.0053	313.3	200.8	164.9
Ray Tracing	0.0014	0.0105	77.6	21.1	21.9

Table 1. Efficacy of point blocking with/without sorting

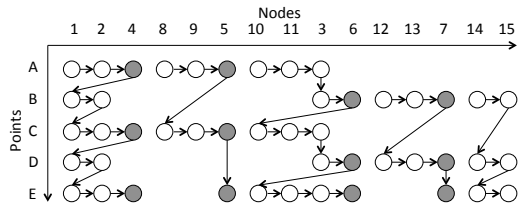


Figure 5. Iteration order for traversal splicing

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. Thus, rather than picking a block of points and following their traversals in lock-step through the entire tree, traversal splicing 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 5 shows the effects that traversal splicing has on the iteration space from Figure 3(b), with the iterations that visit the splice nodes (④, ⑤, ⑥ and ⑦) filled in. 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), its traversal 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 on the order of the number of nodes between splice nodes. As long as the splice nodes are not too far apart, 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).

Two points are worth noting regarding the behavior of traversal splicing. First, the locality effects are the mirror image of point blocking, where the points enjoy good locality and the tree nodes miss once per point block. Second, combining point blocking and traversal splicing does not provide any additional locality benefits. As long as the point blocks are not too large and the splice nodes are not too far apart, we will still suffer only cold misses on the tree nodes and misses on points once per partial traversal (alternately, if we interchange the loops, we will still suffer cold misses on points and misses on tree nodes once per point block). So why perform traversal splicing at all?

The answer lies in the irregularity of the traversals. Section 2.2 already explained how unsorted points can lead to more misses when performing point blocking. A lack of regularity between traversals can impact the locality of traversal splicing, as well. Consider the accesses to node ⑥ made by points B and D in Figure 5. Because the input is unsorted, C performs its partial traversal between the partial traversals for B and D, accessing nodes ⑩ and ⑪, leading to a larger reuse distance for ⑥ than if B and D performed their partial traversals consecutively. This problem, too, can be ameliorated with sorted points. Unfortunately, as discussed in Section 2.1, point sorting requires algorithmic knowledge and is not suited for a general transformation.

To overcome this problem, 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 5. 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 3(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. As a result, if we combine point blocking and traversal splicing with this dynamic reordering transformation, we effectively get the behavior of point blocking with sorted points: point blocks that have high effective block size. Section 6 quantifies how the resorting helps with block density, and demonstrates that performing traversal splicing with reordering can yield results that are, in some cases, competitive with manually-optimized, hand-sorted applications.

Note that from now on, *traversal splicing* refers to a transformation that combines traversal splicing with the point blocking optimization.

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. In applications such as nearest-neighbor (NN), the traversal order is not fixed. For example, at a given node, some points may visit the node's left child before its right, while other points visit its right child before its left.

This situation 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 this new scenario, 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.

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. Because different points are at different splice nodes, the reordering can be more effective at identifying points with overlapping traversals. 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.

3.2 Correctness

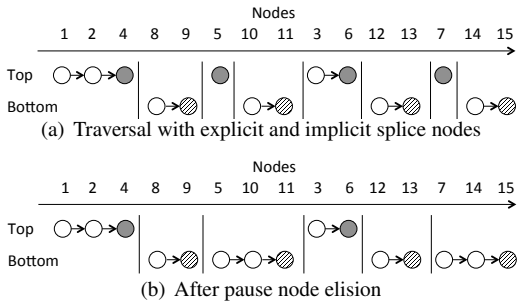
Traversal splicing performs a comprehensive restructuring of a program's access patterns to a tree. Care must be taken, therefore, to ensure that the restructuring does not violate any dependences. To explain the criteria governing the correctness of traversal splicing, we appeal to the iteration space 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 2, 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 (though not strictly necessary): 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.

3.3 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.



(a) Traversal with explicit and implicit splice nodes

(b) After pause node elision

Figure 6. Explicit/implicit splice nodes and top/bottom phases

2. If splice nodes are too deep in the tree, then points are likely to diverge or be truncated before encountering their first splice node. This will result in poor behavior when traversing portions of the tree "above" the splice nodes, and leave less time for the reordering of points to have any effect.
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 large, allowing too much divergence.

Thus, 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 that in Section 3.3, 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 5) 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

```
1 void recurse(Point p, KDNode n) {
2   if (cond) {
3     recurse(p, n.leftChild);
4   } else {
5     recurse(p, n.leftChild);
6     recurse(p, n.rightChild);
7   }
8 }
```

(a) Pseudo-tail recursion

```
1 void recurse(Point p, KDNode n) {
2   if (cond) {
3     recurse(p, n.rightChild);
4     recurse(p, n.leftChild);
5   } else {
6     recurse(p, n.leftChild);
7     recurse(p, n.rightChild);
8   }
9 }
```

(b) Order inferred from node

Figure 7. Levels of optimization

resulting decomposition of a traversal over the tree in Figure 3(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 unique sequence of recursive calls (*i.e.*, a unique 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.

Figure 7(a) shows an example of a pseudo-tail recursive method. Here the two possible call sets (traversal orders) are {leftChild (line 3)}, and {leftChild, rightChild (lines 5-6)}, and which call set a point will take is decided before the traversal is started.

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. We note that 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 thus only needs to maintain which call set it is using at a given level. Truncated points (that do not make it down to a splice node in a given phase) can simply be stored at whichever node they were truncated.

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. This means that, 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. Figure 7(b) shows an example of an algorithm from which order can be inferred. There are two call sets: {leftChild, rightChild (lines 3-4)}, and {rightChild, leftChild (lines 6-7)}, and hence two phases. 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 Figure 1.

4.4 Splice node elision

As a final optimization, recall that the main purpose of traversal splicing is to increase effective block size by reordering the points after each partial traversal. The reordering is informed by the different traversals taken by points leading up to a splice node. However, if the partial traversal is short, it is unlikely that points will have behaved significantly differently, and there is not much new information to drive sorting. This is especially true for certain top phases. Consider the top phase starting at splice node ⑤ in Figure 6(a); the partial traversal 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). Section 5.4 discusses our strategy for splice node elision.

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.

5.1 Recursion unrolling

Top phases need to save additional state so that traversals can be resumed after implicit spice nodes. This is realized by a *transformed* recursive method that saves points and call set id (if applicable) into the tree. The transformed recursive method performs 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.

When resuming a traversal after either an implicit or an explicit splice node, we must determine at which node the next phase starts

from. To facilitate finding the start node, we map the top of the tree up to the children of splice nodes, and save the nodes into an array based on a heap ordering of the tree as in Figure 3(a), before any traversal is started. Non-existent nodes are marked null in the array.

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. The top phase is 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, which basically expands the remaining top phases and appends a bottom phase after each top phase. The depth of the nodes at which to resume, and the phase number is passed as arguments to the top phase (for a bottom phase, the depth argument will always be $D + 1$). For our running example ($D = 3$), the order of phases is T1-0 (top phase at depth 1, traversal phase 0), B (bottom phase), T3-1, B, T2-1, B, T3-1, 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.

At each top phase, we gather all the points that 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. For example at T3-1, we will gather points that have been paused or truncated at nodes ④–⑮, and resume at nodes ④–⑦. At T2-1 we will gather points at nodes ②–⑮, and resume at nodes ② and ③. For each bottom phase, all points that are paused at explicit splice nodes will traverse the appropriate subtrees below their splice node, as determined by their call set id. For inferred order algorithms, as in Figure 7(b), the call set id is unnecessary.

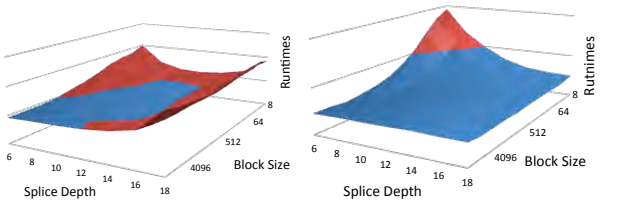
5.2 Dynamic sorting

Recall that the main motivation for applying splicing is to get the enhanced locality benefits of dynamic sorting at splice nodes. In our implementation the dynamic sorting comes naturally, as points are reordered during top phases. Intuitively, each point tracks which explicit splice node it visited most recently. In each phase, all the points that are resuming their traversals at a given node will be reordered according to the explicit splice node they visited last, using a stable sort. Note that we do not need to perform sorting for bottom phases that resume immediately after explicit splice nodes: at the beginning of each bottom phase, all the points at a given node have clearly visited the same explicit splice node last.

5.3 Automatic transformation

We developed a transformation framework, called *TreeSplicer*, that can apply splicing automatically. TreeSplicer is written as a series of passes in the JastAdd framework [9]. TreeSplicer performs point blocking as described in [14]. It then identifies if splicing can be applied by examining if the recursive method is pseudo-tail recursive, and if so which level of optimization is possible. We do not transform non pseudo-tail recursive codes. The optimization level is decided by producing a matrix of all call sets in the recursive method. If there are no conflicting traversals in all traversal phases, the node inferred optimization can be used. 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.

A SpliceSet class is synthesized with code to perform the recursive unrolling of the traversal, and execute top and bottom phases in the right order. A copy of the recursive method and all intermediary methods (non-recursive methods from which the recursive method is first called) leading to the recursive method are made. The recursive method copy is transformed for the top phase: recursive calls



(a) Nearest Neighbor (b) Point Correlation

Figure 8. Runtime with varying block sizes and splice depths

other than the first call in each call set are removed, and points are saved at tree nodes when they are truncated, or reach a splice node. Each intermediary method is split into prologue and epilogues, and all prologues are executed before the first top phase, and all epilogues are executed after the last bottom phase. Any local variables that are defined in the prologue(s) and used in the epilogue(s) are saved in additional global space allocated per point.

5.4 Autotuning

Critical to the performance of splicing is selecting a good block size, B, and splice depth, D. These parameters are dependent on both machine (*e.g.*, L1, L2 cache size) and algorithmic characteristics (*e.g.*, effective block size, average traversal size). Jo and Kulkarni proposed an autotuning technique 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 [14]. A similar approach is infeasible for splicing, because splicing requires seeing a large set of points to perform its dynamic reordering. Splicing over a small set of points can give different results than splicing with the full set.

To explore how B and D impact performance, we measured the runtime of two benchmarks with varying block sizes and splice depths as shown in Figure 8. Our results suggest that there is a large range of parameters for which the performance is good; splicing is not very sensitive to small changes in B or 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, and the sensitivity study shows that a good splice depth performs close to the optimal splice depth. We use the autotuning technique of [14] to choose the block size, and record the average reach of all points in the test blocks. We set the splice depth 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 the splice depth.

For well sorted points, splicing is unlikely to attain locality benefits, and will only result in additional overhead. Let us define *convergence* as the ratio of overlap between consecutive traversals. It can be calculated by recording two traversals, and dividing the size of the intersection of the traversals by the average of two traversal sizes. We observed empirically that sorted points have convergence over 0.5, meaning the traversals of two consecutive points overlap for half their nodes, whereas unsorted points have convergence less than 0.5. Just as we tracked average reach during autotuning, we also profile convergence, and apply splicing only if the convergence is less than 0.5.

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 each of our benchmarks, and finally show that splicing is often competitive with manual optimizations that exploit semantic knowledge.

6.1 Evaluation methodology

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

- **Base**: the baseline described for each benchmark below.
- **Blocking**: automatic point blocking as described in [14].
- **Splicing**: automatic traversal splicing as described in Section 5.

Note that our baseline benchmarks are true baselines: no application-specific *a priori* sorting is performed. Descriptions of the four benchmarks follow.

Barnes-Hut (BH) is a scientific kernel for performing N-body simulation [2]. 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 [16] with one million randomly generated bodies.

Point correlation (PC) is described in detail in Section 2. PC is also an inferred order algorithm with a single call set. We use an input of one million randomly generated points in 3 dimensions, with a correlation radius, r, chosen so that the average correlation covers 0.37% of the points.

Nearest neighbor (NN) search is an optimization problem that arises often in data-mining, and involves finding closest points in metric spaces. Like PC, NN is also accelerated by a kd-tree, by pruning nodes that cannot be closer than the current closest find [12]. NN is an inferred order algorithm, but has *two* call sets. We use a training set and test set of one million points each, randomly generated in a 7-dimensional space, and for each point in the test set, find the nearest neighbor in the training set.

Raytracing (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.* [28]. Ray tracing is the most general benchmark we tackle, as it is *not* an inferred order algorithm and hence we must track each call set explicitly. However, it is pseudo-tail recursive. The input is a randomly generated scene with four million triangles. We rendered a scene with 1024×1024 rays which are processed in random order to simulate the effect of unsorted points.

Platforms We evaluate our benchmarks on two systems with different cache configurations.

- The **Opteron** system runs Linux 2.6.24 and contains two dual-core AMD Opteron 2222 chips in SMP configuration. Each chip has 128K L1 data cache per core and 1M L2 cache per core. We present results up to 4 threads.
- The **Niagara** system runs SunOS 5.10 and contains two 8-core UltraSPARC T2 chips in SMP configuration. Each chip has 8K L1 data cache per core and 4M shared L2 cache. We present results up to 16 threads.[1]
- The **Opteron II** system runs Linux 2.6.32 and contains four twelve-core AMD Opteron 6176 chips in SMP configuration. Each chip has 64K L1 data cache per core, 512K L2 cache per core, and two 6M shared L3 caches. We present results up to 48 threads.

TreeSplicer takes sequential code as input, and outputs sequential code. To test our benchmarks on multicores, we manually parallelized the three versions of each benchmark, with Java threads.

[1] The Niagara supports up to 8-way multithreading. We do not evaluate multithreaded configurations, as multithreading hides latency and both obscures the benefits of locality optimizations and reduces the need for them.

Benchmark	Nor. Eff. Block Size		Heap Size (MB)		
	Baseline	Splicing	Baseline	Splicing	% increase
BH	0.0044	0.0088	220	314	42.7
PC	0.0067	0.1429	197	206	4.6
NN	0.0022	0.0048	408	436	6.9
RT	0.0014	0.0060	881	1656	88.0

Table 2. Effective block size and heap increase on Opteron

Base and **Blocking** were parallelized by processing multiple points or blocks in parallel, and load balancing was applied with work stealing. **Splicing** was parallelized by statically and uniformly distributing the points among threads, with each thread applying splicing to its portion of points. The benchmarks were written in Java and executed on the HotSpot VM 1.7 for the Opteron II and HotSpot VM 1.6 for the Opteron and Niagara, all with 12GB heap. To account for the effects of JIT compilation, each configuration was run 10 times, and the average of the latter 7 runs was recorded. For each benchmark, only the traversal phases were timed.

6.2 Effective block size and heap usage

Table 2 shows the normalized effective block size, and the memory footprint for the baseline and splicing, measured on the Opteron system. It can be seen that splicing substantially improves the normalized effective block size, and for NN, is nearly as effective as *a priori* sorting (see Table 1). The increased memory is modest, less than 10% for PC and NN, which exploit the inferred order optimization. BH also has an inferred order, but uses more memory due to local variables in intermediary methods that must be saved between the prologue and epilogue for *all* points. RT has the largest increase due to call set id stacks, which must be tracked for each point, and also local variables that must be saved between the prologue and epilogue.

6.3 Performance improvements

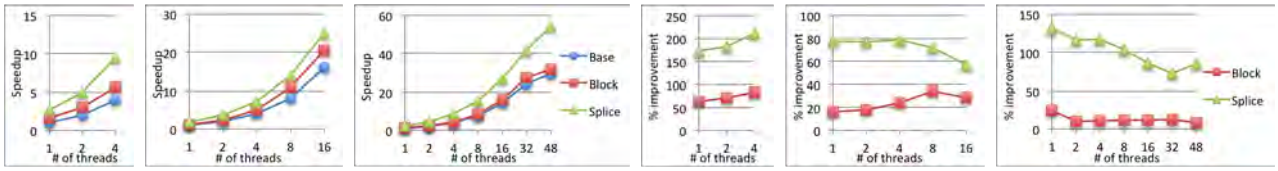
Figures 9–12 show results for BT, PC, NN and RT. For each figure, subfigure (a) shows shows speedups of all versions compared to the serial baseline on the Opteron system; (b), (c) show the same for the Niagara and Opteron II system. (d) shows % improvement of **Blocking** and **Splicing** compared to the parallel baseline on the Opteron; and (e), (f) show the same for the Niagara and Opteron II.

Barnes-Hut (Figure 9): **Blocking** is able to attain maximum improvements of 83.5% , 34.3% and 24.4% over **Base**, on the Opteron, Niagara and Opteron II respectively. **Splicing**, with its dynamic sorting, tops this with maximum improvements of 211.5%, 78.5% and 131.4% over **Base**, and 69.7%, 53.1% and 95.7% over **Blocking**.

Point correlation (Figure 10): **Blocking** gets significant improvements of up to 136.0%, 126.2% and 130.6% over **Base**. However **Splicing** is able to do much better at improvements of up to 393.0%, 275.1% and 379.4% over **Base**, and 108.9%, 93.2% and 183.5% over **Blocking**.

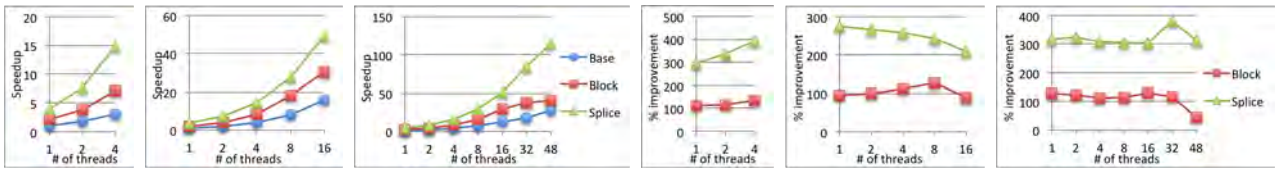
Nearest neighbor (Figure 11): **Blocking** does poorly with maximum improvements of 21.9% , 10.4% and 11.4% over **Base**, due to divergence of points from two call sets, and very small effective block size. **Splicing** improves the effective block size, and results in improvements of up to 150.6%, 134.2% and 130.0% over **Base**, and 105.6% , 112.1% and 106.5% over **Blocking**.

Ray tracing (Figure 12): For RT the divergence of the points is worse enough, and the average traversal size is small enough that **Blocking** does *worse* than **Base** by 7.7% on the Niagara and 4.0% on the Opteron II. The maximum improvement of **Blocking** over **Base** on the Opteron is 33.3%. **Splicing** is able to turn things around: it obtains improvements on the Niagara and Opteron of up to 130.3%, 18.4% and 55.1% over **Base**.



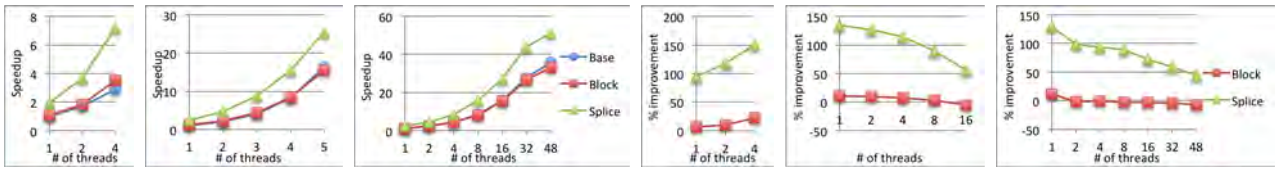
(a) Opteron (b) Niagara (c) Opteron II (d) Opteron (e) Niagara (f) Opteron II

Figure 9. Results for Barnes-Hut: (a)-(c) are speedup vs serial, (d)-(f) are % improvement vs parallel



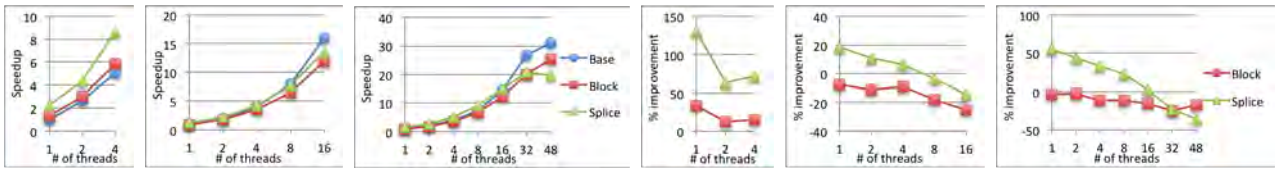
(a) Opteron (b) Niagara (c) Opteron II (d) Opteron (e) Niagara (f) Opteron II

Figure 10. Results for Point Correlation: (a)-(c) are speedup vs serial, (d)-(f) are % improvement vs parallel



(a) Opteron (b) Niagara (c) Opteron II (d) Opteron (e) Niagara (f) Opteron II

Figure 11. Results for Nearest Neighbor: (a)-(c) are speedup vs serial, (d)-(f) are % improvement vs parallel



(a) Opteron (b) Niagara (c) Opteron II (d) Opteron (e) Niagara (f) Opteron II

Figure 12. Results for Ray Tracing: (a)-(c) are speedup vs serial, (d)-(f) are % improvement vs parallel

Several trends are evident in these results. First, 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 from splicing decline as the number of threads increase on the Niagara and Opteron II. This is due to two factors. First, splicing'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. Second, autotuning is currently done sequentially and limits speedup according to Amdahl's law, for both blocking and splicing which have autotuning phases. The improvements from splicing *increase* with the number of threads on the Opteron. We speculate that this is due to bus saturation; because splicing reduces cache misses, it reduces bus pressure, resulting in comparatively more improvement compared to the baseline which does not scale. This effect is accentuated for the Opteron which has less bus bandwidth, overcoming the two factors discussed above, resulting in net improvement as the number of threads increase.

Table 3 shows performance counter results on the Opteron II collected with PAPI [8], which demonstrate that point blocking and traversal splicing have much lower CPI and L2 miss rates, indicating that the improvements are indeed from better locality. Traversal splicing actually has fewer instructions than point blocking as its dynamic sorting reduces the number of nodes each block must traverse. RT has a small average traversal size and hence low CPI even for the baseline, and the increased instruction overhead is high due to many intermediary methods leading to the recursive method, which is why we see less improvement from splicing compared to the other benchmarks.

6.4 Comparison to manual optimization (sorting)

In the previous section, we showed that splicing can attain substantial improvements over a 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? Figure 13 compares the performance of two approaches to the baseline: splicing, and a manually-optimized implementation that performs point sorting and blocking. The results are for a single thread on both systems. Sorting time is included for NN, where sorting has additional overhead because the points are different from the entities used to build the tree. Sorting is trivial for the other benchmarks as points can be read off the tree (BH, PC), or points are inherently sorted (RT).

While sorting + blocking does best in general, for many benchmarks splicing is competitive. In particular for NN, splicing, with its dynamic sorting, is 2% better than the manually optimized version, alluding to the difficulty of doing *a priori* sorting for codes with more than one call set. Averaged across all benchmarks and systems (geometric mean), our automatically transformed code is only 23.6% worse than the hand-optimized implementation.

Benchmark	Instructions (billions)			CPI			L2 miss rate		
	Baseline	Blocking	Splicing	Baseline	Blocking	Splicing	Baseline	Blocking	Splicing
Barnes-Hut	65	93	74	4.737	2.536	1.702	0.4998	0.4277	0.2243
Point Correlation	500	562	552	3.957	1.683	0.955	0.4702	0.3533	0.0822
Nearest Neighbor	295	476	396	3.142	2.016	1.115	0.6736	0.4808	0.1807
Ray Tracing	47	87	70	2.421	1.404	1.056	0.4545	0.3043	0.1807

Table 3. Performance counter results on Opteron II

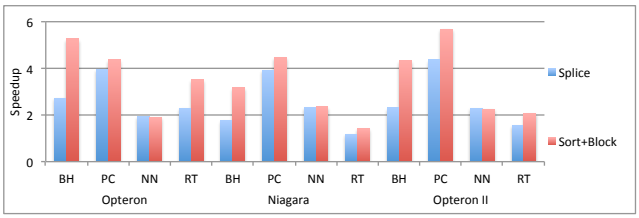


Figure 13. Speedup of splicing and sorting + blocking over baseline for 1 thread

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 [7, 20, 26], 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. Singh *et al.* order the points in Barnes-Hut according to their position in the oct-tree [25], while Amor *et al.* use space-filling curves [1]. Mansson *et al.* perform a similar optimization for reflected rays in ray tracing [18].

There have been a few application-specific approaches similar to traversal splicing. Pharr *et al.* improve the memory coherence of ray tracing by breaking a scene into "voxels." 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 [21]. 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 [22]. 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 [11, 24], or parallelization [10, 23]. These approaches are complementary to ours. Indeed, 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. These have revolved around modifying memory allocation to control data layout, either automatically [17], through leveraging application semantics or programmer annotations [4, 5, 27], or by performing relayout during garbage collection [6]. 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 point correlation. 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. Traversal splicing is based on 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 in some cases, applying traversal splicing to a traversal algorithm, even in the absence of any hand optimization, is competitive with manually-transformed, locality-enhanced implementations of the same algorithm.

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

The authors would like to thank Bruce Walter for providing the Raytracing benchmark code. We would also like to thank Vijay Pai and his students for providing support for machines on which our tests were conducted.

References

[1] 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.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.