Enhancing Locality for Recursive Traversals of Recursive Structures

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

Abstract

While there has been decades of work on developing automatic, locality-enhancing transformations for regular programs that operate over dense matrices and arrays, there has been little investigation of such transformations for irregular programs, which operate over pointer-based data structures such as graphs, trees and lists. In this paper, we argue that, for a class of irregular applications we call *traversal* codes, there exists substantial data reuse and hence opportunity for locality exploitation.

We develop a novel optimization called *point blocking*, inspired by the classic tiling loop transformation, and show that it can substantially enhance *temporal* locality in traversal codes. We then present a transformation and optimization framework called *TreeTiler* that automatically detects opportunities for applying point blocking and applies the transformation. TreeTiler uses *autotuning* techniques to determine appropriate parameters for the transformation. For a series of traversal algorithms drawn from real-world applications, we show that TreeTiler is able to deliver performance improvements of up to 245% over an optimized (but non-transformed) parallel baseline, and in several cases, significantly better scalability.

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

General Terms Languages

Keywords locality transformations, irregular programs, tree traversals

1. Introduction

It has long been understood that locality is a crucial factor in delivering high performance scientific applications. Over the past several decades, there has been substantial work on automatically transforming *regular* programs, which operate over dense matrices and arrays, to enhance locality. These investigations have led to the creation of catalogs of transformations and techniques to determine when those transformations are legal and effective [16]. In contrast, there has been relatively little attention paid to locality in *irregular* programs, which operate over pointer-based structures such as trees and graphs. While there have been various techniques and transformations proposed for enhancing the locality of specific irregular applications [2, 22, 24, 29], general approaches to improving the locality of broad classes of irregular applications are few and far between.

This lack of progress is unsurprising. Pointer-based data structures are highly dynamic and the resulting memory-access patterns of applications that use them are highly input-dependent and unpredictable. As a result, the standard techniques for reasoning about locality in regular applications are simply inapplicable[1].

The apparent lack of structure in irregular programs can be misleading. While the particular set of concrete memory accesses may exhibit little regularity, at an abstract level there are organizing principles governing these accesses, such as the topology of the irregular data structure, or the nature of operations on that data structure. Recent work by Pingali *et al.* has suggested that there may, indeed, be significant structure latent in irregular applications [25]. Can this structure be exploited to transform irregular applications so as to enhance locality?

In this paper, we focus on enhancing and exploiting *temporal* locality in algorithms that perform repeated traversals of recursive structures, such as trees, DAGs and graphs. Such

[1] While there has been progress, in the form of complex compiler analyses like *shape analysis* [11, 27], in discerning properties of irregular data structures (primarily, their topology), these techniques have mostly been put to ends such as verification and parallelization, rather than locality enhancement.

applications are widespread; examples include scientific algorithms such as Barnes-Hut [3], graphics algorithms such as bounding volume hierarchies [34] and Lightcuts [36], and data mining algorithms such as nearest neighbor and point correlation [12]. The goal of each of these algorithms is to compute a value (force, illumination, etc.) for each of a set of entities (bodies, rays, etc.). This computation is performed by constructing a tree-based acceleration structure and then traversing that structure for each entity to compute the desired value. In other words, these algorithms perform repeated series of tree traversals.

The tree traversals performed by the aforementioned algorithms are highly irregular in nature. This is because the structure of the tree is determined primarily by the input data and because the actual layout of the tree in memory is unpredictable. Nevertheless, the trees constructed in these algorithms are traversed numerous times, leading to significant data reuse. Any time there is data reuse, there may be an opportunity to exploit temporal locality.

By drawing an analogy with loop transformations in regular programs, where *loop tiling* has proved to be an effective technique to improve locality in matrix codes, we develop a novel, locality-enhancing transformation for tree traversal codes that we call *point blocking*. Because point blocking can be applied to any parallelizable tree traversal code, it is a general transformation, and can be effectively employed in all the applications mentioned previously.

We then describe *TreeTiler*, a compiler framework that automatically identifies regions of programs where data reuse implies that point blocking might be successfully applied. In regular programs, data reuse often arises in nested loops that manipulate arrays and matrices, and can be readily identified. In irregular programs, in contrast, data reuse is often masked by pointer-manipulation operations. TreeTiler identifies code where point blocking might be performed by looking for *recursive traversals of recursive structures*. If point blocking is legal for such a traversal, TreeTiler automatically performs the transformation.

Point blocking, like loop tiling, requires that optimization parameters be carefully tuned to match both the application and the architecture. *Autotuning* has emerged as a popular approach to parameter selection as it can select optimization parameters for a particular execution scenario without programmer intervention [31, 33, 37], a necessity for any automated transformation framework. Because irregular programs are highly input-dependent, TreeTiler uses run-time profiling to guide its selection of parameters for point-blocking.

Contributions

The contributions of this paper are threefold:

1. We present an abstract model of tree traversal codes that allows reasoning about locality effects. We then describe a novel transformation, point blocking, that applies to recursive traversal of recursive structures, such as tree traversals (Section 2).
2. We develop TreeTiler, a compiler that identifies opportunities for applying point blocking and automatically performs the transformation (Section 3).
3. We implement two autotuners that use run-time profiling to automatically tune the parameters of a point-blocked application (Section 4).

In Section 5, we evaluate the effectiveness of point blocking, and the TreeTiler transformation and tuning framework, on a suite of five applications that perform tree traversals. The automatically transformed applications achieve performance improvements of up to 245% over hand-optimized parallel baselines that do not use point-blocking. Further, TreeTiler's autotuning is able to select transformation parameters that are competitive with hand-tuned transformations. For several benchmarks, the locality benefits of point blocking also result in significantly greater scalability.

2. Transformations for tree-traversal codes

In this section, we begin by discussing some background on applications that perform recursive traversals over recursive data structure. We next describe an abstract model for reasoning about the locality properties of such applications. Finally, we present the *point blocking* optimization, and discuss its locality effects in relation to our abstract model.

2.1 Background

As discussed in the introduction, we are interested in applications that perform repeated traversals of recursive structures such as trees, DAGs and graphs. Because these repeated traversals each access the same data structure, there is an abundance of data reuse, and hence locality, to be exploited.

These applications all follow the same general pattern. To explain this pattern, we will make reference to perhaps the canonical tree-traversal algorithm, Barnes-Hut [3], whose pseudocode is given in Figure 1[2]. The outer loop of a traversal code iterates over a set of *entities* or *points*; in Barnes-Hut, these are the bodies in space (line 1). For each point, a recursive structure, the *environment* is traversed; in Barnes-Hut, the environment is an oct-tree built over the entities (line 2). This traversal is performed recursively: at each node in the environment, a check is made to see if the traversal should be stopped (line 8) or whether it should continue (lines 11–14). Because the traversal is recursive, it explores the data structure in depth-first order.

Simple locality-enhancing transformations Because the oct-tree in Barnes-Hut is a highly dynamic data structure, exploiting locality in the traversals is difficult. However, as

[2] The full Barnes-Hut algorithm consists of several phases; we concentrate on the force computation phase, which is both the most time-consuming phase, and the phase with the computational structure we are interested in.

```
 1 Set<Point> points = /* entities */
 2 OctTreeCell root = /* environment */
 3 foreach (Point p : points) {
 4   Recurse(p, root);
 5 }
 6
 7 void Recurse(Point p, OctTreeCell c) {
 8   if (farEnough(p, c.cofm) || c.isLeaf) {
 9     updateContribution(p, c.cofm);
10   } else {
11     foreach (OctTreeCell child : c.children) {
12       if (child != null)
13         Recurse(p, child);
14     }
15   }
16 }
```

Figure 1. Force computation algorithm for Barnes-Hut

# Objects	Traversal size (Bytes)	L2 miss rate (%)	% Improvement in cycles over un-optimized
10000	63,944	21.61	67.3
100000	108,656	44.97	45.9
1000000	139,616	55.30	26.4

Table 1. Efficacy of sorting optimization for various traversal sizes

the same tree is traversed by each point (the outer loop in Figure 1), there is significant data reuse. Points that are nearby in space are likely to perform very similar traversals of the oct-tree, visiting the same set of tree nodes. Thus, if these traversals are performed consecutively, the oct-tree nodes visited during the first traversal are likely to remain in cache during the second traversal, exploiting temporal locality.

Such a locality-exploiting order of traversals can be arranged by processing the points according to their geometric position (*e.g.*, with a space-filling curve), so that adjacent points in the sorted order are nearby geometrically [2, 29]; we use this optimization in the baseline we use in the evaluation of Section 5. Though the optimization has only been applied to Barnes-Hut in the literature, we note that analogous transformations can be applied to any traversal code: if the points are sorted to maximize the overlap between consecutive traversals, locality can be improved.

This optimization loses its effectiveness as the traversal sizes get larger. With a sufficiently large traversal, the least recently visited nodes of the oct-tree will be evicted from cache, and hence when the next point is processed those nodes will have to be brought back in to cache, incurring additional misses. Table 1 shows, for several tree sizes, the average traversal size, the L2 miss rate of an optimized implementation, and the % improvement in cycles over an unoptimized implementation. The test system is a dual-core Intel Pentium with 32K L1 data cache per core and 1M shared L2 cache. The efficacy of sorting is clear for small sizes: in an input with 10,000 points, the sorting optimization improves runtime by 67%. However, with an input of 1 million points, the sorting optimization has much higher miss rates, and only improves runtime by 26% compared to the un-

```
Set<Point> points = /* entities in algorithm */
Set<Point> objects = /* environment objects */
OctTreeCell root = buildTreeAndComputeCofM(objects);
foreach (Point p : points) {
  foreach (OctTreeCell c : traverse(root, p)) {
    if (farEnough(p, c.cofm) || c.isLeaf) {
      updateContribution(p, c.cofm);
    }
  }
}
```

Figure 2. Abstract algorithm for tree-traversal

optimized version. Clearly, a more sophisticated optimization is necessary to continue exploiting locality as traversal sizes get larger.

2.2 An abstract model

Reasoning about locality in codes that traverse recursive structures is difficult for a number of reasons. First, unlike in regular applications, the structure of the key data structures is highly input-dependent. The oct-tree generated in Barnes-Hut is dependent on the particular locations of the points in the system. Furthermore, the data structures are dynamically allocated, and hence can be scattered throughout memory. Finally, the traversals are not uniform; a traversal can be truncated (*e.g.*, due to the distance check in line 8 of Figure 1), and traversals for two different points are not necessarily similar.

However, we can still reason about locality by considering the behavior of a traversal algorithm in a more abstract sense. Rather than viewing a traversal as a recursive, depth-first walk of a data structure, we can instead visualize the traversal in terms of the actual nodes touched. Fundamentally, processing a single point requires accessing some sequence of tree nodes. The particular arrangement within the tree of those nodes is irrelevant; all that matters is the ultimate sequence in which the nodes are touched. If we imagine that there is an oracle function `traverse` that generates the sequence of nodes accessed while processing a particular point, we can rewrite the code of Figure 1 as shown in Figure 2. In other words, we can view the algorithm as a simple, doubly-nested loop. Notably, *for the purposes of locality, the behavior of the original Barnes-Hut code is equivalent to the abstract algorithm.* All that matters is the sequence of accesses; the additional computations required to determine whether to continue a traversal or not do not affect locality. Thus, the sequences of memory accesses for the code in Figure 1 and Figure 2 are identical.

Recursive traversals as outer products This abstract algorithm provides insight into why sorting the points (as discussed in Section 2.1) is useful for locality. Consider the behavior of two consecutive points, p_1 and p_2. In the unsorted algorithm, there is little overlap between $traverse(p_1)$ and $traverse(p_2)$. Most of the inner-loop accesses for the p_2 iteration will result in cache misses. However, sorting the

```
Point p[n] = /* entities */
OctTreeCell c[m] = /* traversal */
for (int i = 0; i < n; i++)
  for (int j = 0; j < m; j++)
    Update(p[i], c[j]); //A[i][j] = p[i]*c[j]

void Update(Point p, OctTreeCell c) {
  if (farEnough(p, c.cofm) || c.isLeaf)
    updateContribution(p, c.cofm);
}
```

Figure 3. Traversals as outer product

points such that consecutive points have similar traversals will result in cache hits.

When the points are sorted, the variability between consecutive traversals will be a fairly small second-order effect, so we can simply consider consecutive traversals in the sorted case to be the same. This approximation lets us further simplify the abstract algorithm. The outer loop iterates over a vector (of points) and, for each point, the inner loop iterates over a vector (containing the nodes of the traversal). If there are n points, and the average traversal is m nodes, then this is an $O(mn)$ algorithm with an access pattern equivalent to an $m \times n$ outer product. Figure 3 demonstrates this correspondence, showing how a tree traversal is analogous to the outer product of a vector p and a vector c.

Note that this model elucidates why the efficacy of the sorting optimization decreases as traversals get larger. As long as the average traversal of size m fits in cache, we incur only cold misses on the traversal vector. However, as soon as m exceeds cache, an LRU replacement policy will cause each access to the traversal vector to miss.

2.3 Point blocking

Given the abstract, outer-product model of traversal codes described above, several analogs of classical loop transformation techniques become apparent. For example, loop interchange would place the traversal loop on the outside, with the point loop on the inside. This corresponds to choosing a node of the recursive data structure, then processing each point that must interact with it. Makino proposed a variation of this transformation for Barnes-Hut [21], to facilitate vectorization. However, we note that, just as the original code suffers from poor locality if the traversal vector exceeds cache, the loop-interchanged code will suffer from poor locality if the point vector exceeds cache. For large inputs, this is likely[3].

While loop interchange may not produce an effective implementation of a traversal code, loop tiling holds promise. In particular, we propose tiling the point vector, which produces the code seen in Figure 4. Essentially, this code breaks the points into blocks of size B. For each block, each node of the recursive data structure is chosen, then each point in the block is processed for the chosen node. If B is chosen correctly, the points in a block will never leave cache. Further, regardless of how large the traversal is, each node of the traversal will only incur a cache miss once per block[4]. We call this optimization *point blocking*. Section 3 discusses how to realize point blocking in actual traversal codes (rather than the abstract code) and discusses sufficient conditions for its legality.

[3] In fact, for Barnes-Hut, the point vector has n elements, while the traversal vector has $O(\log n)$ elements, so the interchanged code is more likely to suffer cache misses than the original code.

```
Point p[n] = /* entities */
OctTreeCell c[m] = /* traversal */
for (int ii = 0; ii < n; ii += B) {
  for (int j = 0; j < m; j++)
    for (int i = ii; i < ii + B; i++)
      Update(p[i], c[j]); //A[i][j] = p[i]*c[j]
}
```

Figure 4. Point blocked traversal

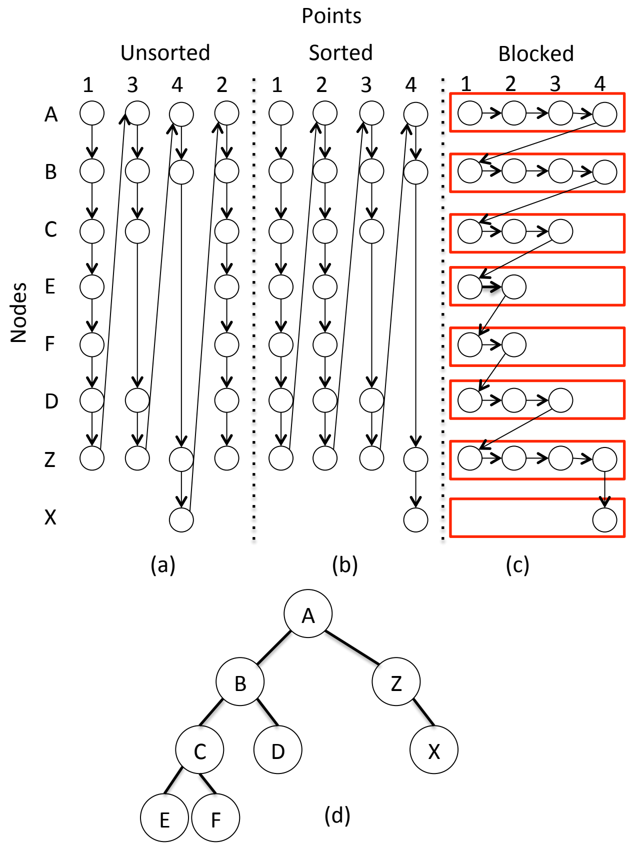


Figure 5. Traversal order of a sample tree

Point blocking example

A simple example will help elucidate the process of performing point blocking; we will explain both the conceptual behavior of the transformation, and illustrate how the transformation is applied to code. We take the Barnes-Hut

[4] In a traversal of a cyclic structure, certain nodes may be visited multiple times, and may incur misses each time.

code of Figure 1 and the sample binary (for simplicity) tree of Figure 5(d). Borrowing from the literature on loop transformations in regular programs, and using our analogy of tree traversal codes to vector outer products, we can visualize our transformations in a two dimensional iteration space. Figure 5(a) shows the original (unsorted) iteration space for four points. Each circle in the iteration space represents one node of the data structure being visited by one point. Each column represents the accesses made by a particular point, while each row represents the accesses made at a particular node of the tree. Note that not every point visits each node, as points' traversals can be different. The arrows show the order in which accesses are made. Point 1 traverses nodes A-F in depth first order, then moves on to node Z, but the termination condition (line 8 of Figure 1) truncates the traversal and prevents it from visiting node X. Next is point 3, and the traversal is truncated at nodes C and Z. For point 4 the traversal is truncated at node B, and finally point 2 has the same traversal as point 1.

As discussed in Section 2.1, it is often possible to *sort* the points, so that points with similar traversals are processed consecutively. This results in the traversal order of Figure 5(b)[5]. Point 2 is processed after point 1, and will enjoy temporal locality from point 1's previous traversal. The problem arises when the traversal outsteps cache. For example, if the cache can only fit 5 nodes, node A will no longer be in cache by the time point 2 comes around to it, even though point 1 accessed the node in its traversal.

Point blocking changes the order as in Figure 5(c). This is for a block size of 4, and the blocks are shown as red rectangles. Now, even if the cache can only fit 5 nodes, points 2, 3 and 4 can exploit temporal locality from point 1's previous access. As in loop tiling, we must take care to keep the point vector in cache by sizing the blocks appropriately. Note that the point blocked code must preserve the traversals of individual points. Points 3 and 4 do not interact with node E in the untransformed code, and hence when node E is visited by the point block, points 3 and 4 should be skipped. A simple point blocked implementation is shown in Figure 6. Nodes are accessed on a per-block basis, and points that need to recurse further are added to a next block (line 19). A complete, more complex example will be discussed next.

3. Automatic transformation with TreeTiler

In the previous sections we have discussed how loop transformations can significantly reduce cache misses in codes that repeatedly traverse trees and other recursive structures. Realizing these transformations is non-trivial because each point may have a different traversal (*i.e.*, each point may require traversing a different portion of the data structure). As long as the differences in traversals between consecutive points are small, the point blocking transformation can be

[5] Changing the order of points does not affect the set of nodes that each point must access.

```
 1  Set<Point> points = /* entities */
 2  OctTreeCell root = /* environment */
 3  Block b = new Block();
 4  foreach (Point p : points) {
 5    b.add(p);
 6    if (b.size == blockSize) {
 7      Recurse(root, b);
 8      b = new Block();
 9    }
10  }  // handle remaining points
11
12  void Recurse(OctTreeCell c, Block b) {
13    Block nextB = new Block();
14    for (int i = 0; i < b.size; i++) {
15      Point p = b.p[i];
16      if (farEnough(p, c.cofm) || c.isLeaf) {
17        updateContribution(p, c.cofm);
18      } else {
19        nextB.add(p);
20      }
21    }
22    if (nextB.size > 0) {
23      foreach (OctTreeCell child : c.children) {
24        if (child != null)
25          Recurse(child, nextB);
26      }
27    }
28  }
```

Figure 6. Point blocked code for Barnes-Hut

effective; nevertheless, we must ensure that these differing behaviors are respected by the transformation. In this section, we describe an analysis and transformation framework, called *TreeTiler*, which can apply the transformations automatically. TreeTiler is written as a series of passes in the JastAdd framework [7], which enables analysis and transformation of Java programs.

TreeTiler consists of several passes, which we describe in more detail in the following sections:

1. *Identifying targets for point blocking.* TreeTiler finds possible transformation opportunities by looking for code that performs repeated traversals of recursive structures. (Section 3.1).
2. *Verifying correctness.* TreeTiler analyzes the dependences in the identified loop to determine whether point blocking can be legally performed. (Section 3.2).
3. *Applying point blocking.* If point blocking is legal, TreeTiler automatically performs the transformation. (Section 3.3).

3.1 Identifying opportunities for point blocking

While 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. If an application performs *repeated recursive traversals of a recursive structure*, TreeTiler will identify it as a candidate for point blocking.

Thus, the first step in this phase is to determine whether an algorithm consists of a recursive traversal of a recursive structure. Hereafter, we will use Java terminology and refer

```
1   class OctTreeCell {
2     OctTreeCell [] children;
3     void Recurse(Point p) {
4       if (farEnough(p, cofm) || isLeaf) {
5         updateContribution(p, cofm);
6       } else {
7         foreach (OctTreeCell child : children) {
8           if (child != null)
9             child.Recurse(p);
10        }
11      }
12    }
13}
```

Figure 7. Passing recursive class via implicit argument

to functions as methods, and data structures as classes. We define a recursive class as a class with fields of its own type (which we call its children). This class represents the nodes of the structure being traversed, and the traversal of a point is realized by recursively calling a method on the children of a node. The recursive method has some termination condition dependent on both the point being processed and the current node being traversed. If the termination condition is satisfied, the recursion is stopped, and the traversal proceeds with recursion at a previous method call. Depth-first order is maintained naturally by the program stack.

This doubly recursive structure is illustrated for Barnes-Hut in Figure 1. The node class *OctTreeCell* is a recursive class with fields *children* that are also of type *OctTreeCell*. The method *Recurse* (lines 7-16) is a recursive method that takes a recursive class as an argument. The traversal is realized by calling *Recurse* on the children of a node (line 13). A termination condition stops the recursion if the point is far enough away from the node (line 8).

The algorithmic structure that we want to identify must be a combination of both *recursive method calls* and *recursive structures*. Recognizing each individually is trivial. A recursive method, m, can be recognized by finding a call to itself within a method's body[6]. A recursive structure can be recognized by finding a class, c, with at least one field f of the same class (or superclass).

We must then determine 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). The former was illustrated in Figure 1. Figure 1 could be re-written to Figure 7, where the explicit argument child in line 13 of Figure 1 has changed to the implicit argument in line 9 of Figure 7. Using implicit arguments is common programming style, and TreeTiler handles both cases.

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

Having identified a recursive traversal of a recursive structure, TreeTiler's next goal is to determine if it is repeated. To do so, TreeTiler 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 there is a single path from the enclosing loop to the recursive method, TreeTiler transforms the code as described in Section 3.3. TreeTiler will perform the transformations for every kernel it identifies that is a *repeated recursive traversal of a recursive data structure*.

3.2 Correctness of transformation

As in any loop transformation in regular programs, our transformation must preserve dependences to ensure correctness. We will refer to the iteration space of our simple example, which was depicted in Figure 5.

Note that while the transformed code walks the iteration space in a different order than the original code, a few key aspects of the execution order are preserved. First, for a given point, nodes are visited in the same order. Hence, intra-point dependences (dependences that point "down" in the iteration space) are preserved. Second, if the data structure being traversed is a tree, for a given node, points "visit" the node in the same order. Hence, while not all inter-point dependences are respected by the transformation, intra-node dependences, where values on a node are updated as it is visited by points, are preserved as well.

Other types of dependences are not preserved. For example, if processing a point changes the tree structure, subsequent points' iteration spaces will be affected, and the transformed traversal order may produce different results. Further, if the structure being traversed is not a tree, there may be multiple paths to reach a certain node, or a traversal may access a node multiple times; in these situations, inter-point dependences (even those that are intra-node) may not be preserved by the transformation. Section 3.4 discusses some of the implications of traversals of non-tree data structures.

To handle these situations, TreeTiler performs two checks. First, it ensures that the data structure is not morphed during the traversal (*i.e.*, that the recursive fields are not written to). Second, TreeTiler checks if the traversal is parallelizeable as a conservative guarantee that there are no problematic inter-point dependencies. If both checks pass, then point blocking is legal. Note that parallelizability is a *sufficient* condition for point blocking to be legal, not a necessary one; tree traversals where values on nodes are updated as points visit them can be transformed, but not parallelized.

3.3 Implementation of transformation

Once we have identified the recursive structures that can be transformed correctly, the next step is to realize the transformation efficiently. A generic recursive structure that we have identified will look like Figure 8. Our analysis will find a recursive method associated (explicitly or implicitly) with a recursive class (lines 10-17), and the call path to an en-

```
1   Set<Point> points = /* entities in algorithm */
2   Object o1 = /* something loop invariant */
3   foreach (Point p : points) {
4     Object o2 = /* something loop variant */
5     // do something - prologue
6     Object o3 = recurse(p, o1, o2, root);
7     // do something - epilogue
8   }
9
10  void recurse(Point p, Object o1, Object o2, Node node) {
11    // do something
12    if (cond) {
13      foreach (Node child : node.children) {
14        recurse(p, o1, o2, child);
15      }
16    }
17  }
```

Figure 8. Original generic recursive structure

closing loop (lines 3-8). There can be an arbitrary number of methods in the call path, which may or may not have return values. There can be arbitrary code between each method call on the path to the recursive method (lines 5, 7). And there can be arbitrary arguments passed along methods in the call path. The generic code shows two such arguments, one that is loop variant and another that is loop invariant with regard to the enclosing loop. This distinction is necessary because loop variants will require extra space, proportional to the size of the block. The type of the arguments or return values is irrelevant, they are deemed Object for the sake of illustration. Arbitrary intermediate methods are not shown in Figure 8, and discussed further in Section 3.3.3.

The transformed code of the generic recursive structure is shown in Figure 9. The automatically generated block classes corresponding to the transformed code are discussed in Appendix A. We will now discuss the transformation step by step. We will first explain how TreeTiler operates in the simple case: code such as Barnes-Hut. We will then describe how TreeTiler handles complications in the basic algorithmic pattern: multiple recursive calls within a method, and chains of method calls between the loop and the recursive method.

3.3.1 Basic transformation

The first step in tiling a recursive code is to transform the enclosing loop (the entry to the call path to the recursive method) to be over blocks of points, rather than single points. In the original code, each point is processed by calling *foo* on the root node. In the transformed code, points will be added to a block instead (line 9 of Figure 9); once this block is full, a modified version of *foo* will be called *on the entire block*. Note that this block may contain more than just the point; any loop-variant arguments to *foo* (such as *o2*) are also placed in the block. The block also contains space for the return value of *foo*, as it is also loop variant. Loop invariant arguments (such as *o1*) are passed to *foo* without change.

The original enclosing loop can have arbitrary code before (prologue, line 5 of Figure 8) and after (epilogue, line 7 of Figure 8) the method call. The prologue need not be

```
1   Set<Point> points = /* entities in algorithm */
2   Object o1 = /* something loop invariant */
3   Block b = /* block instance */
4   BlockStack stack = /* stack instance */
5   // autotuning code to be added here
6   foreach (Point p : points) {
7     Object o2 = /* something loop variant */
8     // do something - prologue
9     b.add(p, o2);
10    if (b.size == blockSize) {
11      stack.set[0].block = b;
12      recurse(o1, root, b);
13      for (int i = 0; i < b.size; i++) {
14        Point p = b.p[i];
15        Object o2 = b.o2[i];
16        Object o3 = b.ret_foo[i];
17        // do something - epilogue
18      }
19      b.recycle();
20    }
21  } // handle remaining points
22
23  void recurse(Object o1, Node node,
          BlockStack stack, int level) {
24    BlockSet bset = stack.set[level];
25    Block b = bset.block;
26    Block nextB = bset.nextBlock;
27    nextB.recycle();
28    if (Block.tuning) Block.workDone += b.size;
29    for (int i = 0; i < b.size; i++) {
30      Point p = b.p[i];
31      Object o2 = b.o2[i];
32      // do something
33      if (cond) {
34        nextB.add(p, o2);
35      }
36    }
37    if (nextB.size > 0) {
38      stack.set[level + 1].block = nextB;
39      foreach (Node child : node.children) {
40        recurse(o1, child, stack, level + 1);
41      }
42    }
43  }
```

Figure 9. Transformed generic recursive structure

changed, as in the transformed code the prologue will be executed for each point of the block before the method call. The epilogue however, must execute after the method call, which will not be accomplished simply by leaving it at its original position after a call to add to the block. The epilogue is moved to a new loop that iterates over points of the block (lines 13-18 of Figure 9). This new loop has access to the loop variant arguments, which were added to the block, as well as the return value of *foo*.

The block *b* is recycled after it has been processed (line 19 of Figure 9). Recycling a block, simply sets its size to 0, so that the next invocation of *add* will overwrite previous points. If the last points in the loop cannot fill a block, the partial block is processed as in lines 12-18 of Figure 9.

The next step is to transform the recursive method that performs the traversals. Because the traversals of individual points are different, a block of points must traverse a set of data structure nodes that is the union of the traversals of all the points within the block. If a node from this superset should not interact with a point in the block, that point should be skipped. To ensure that points within a block skip the appropriate nodes, we must somehow track which points should interact with which nodes. In general this would require space per point proportional to the entire traversal, but the depth-first order allows us to use space proportional to the depth of the traversal. This comes from the observation that once a point is skipped for level l, it will also be skipped for all subsequent levels $l + n$ along a depth-first recursion path. Because a given depth-first path through the tree only accesses one tree node per level, we need only keep track of a point's information once at each level of the tree, which results in one block's worth of information per tree level.

Rather than allocate a new block at each level (as in Figure 6), it is more efficient to preallocate a *block stack*, which has a block *set* per level. Each set has a reference to the current block for the level, and allocated space for any block(s) that might be needed for the next level. The block stack and the current level of the traversal, are passed to the recursive method. The method is executed for each point in the current block; recursive invocations in the original code are replaced, and instead add points to the next level's block(s). Once each point has been processed for the current level, the method is called on the next level's block(s).

3.3.2 Handling multiple recursive calls

Interestingly we may need multiple blocks for the next level. This is because in some algorithms, points access children in different orders during their depth-first traversal. For example, this situation arises in the Nearest Neighbor benchmark, and is illustrated in Figure 10. Figure 10(a) shows the original recursive method, and Figure 10(b) shows how it should be transformed. The order in which the children are processed can be either left first then right, or right first then left, depending on the point. The transformed code must honor both orders, by having two next blocks. Points that take the first traversal order are added to the first block, while those that take the second traversal order are added to the second block. At the end of the current level, both next blocks are processed in the appropriate order.

It may seem that we should have a next block for each recursive invocation in the transformed method. For example, in the generic code of Figure 8, where there are an arbitrary number of children, it seems that we might need an arbitrary number of blocks. The crucial difference between the scenario in figure 8 and the scenario of Figure 10 is that the latter has a divergence of control flow dependent on the point's properties. Hence, the traversal orders may differ on a point-by-point basis, and separate blocks are necessary to differentiate between the orders. In the generic scenario, lines 13–15 in Figure 8 will be executed for *all* points in the block, hence it can be replaced with a single block *add* call (line 34 of Figure 9). Lines 13–15 in Figure 8 are moved to lines 39-41 in Figure 9.

TreeTiler decides the number of next blocks required by starting at each recursive method call site within the recursive method body, and expanding the call site to the

```
void recurse(Point p, Node n) {
  boolean cond = // do something dependent on p
  if (cond) {
    recurse(p, n.left);
    recurse(p, n.right);
  } else {
    recurse(p, n.right);
    recurse(p, n.left);
  }
}
```

(a) **Original code**

```
void recurse(Node n, BlockStack stack, int level) {
  BlockSet bset = stack.set[level];
  Block b = bset.block;
  Block nextB = bset.nextBlock;
  Block nextB2 = bset.nextBlock2;
  nextB.recycle();
  nextB2.recycle();
  for (int i = 0; i < b.size; i++) {
    Point p = b.p[i];
    boolean cond = // do something
    if (cond) {
      nextB.add(p);
    } else {
      nextB2.add(p);
    }
  }
  if (nextB.size > 0) {
    stack[level + 1].block = nextB;
    recurse(p, n.left);
    recurse(p, n.right);
  }
  if (nextB2.size > 0) {
    stack[level + 1].block = nextB2;
    recurse(p, n.right);
    recurse(p, n.left);
  }
}
```

(b) **Transformed code**

Figure 10. Traversing different orders of children

largest control flow block that is control independent of the point. Each expanded call site requires its own next block. The required number of next blocks is synthesized in the block set class, and each expanded call site is replaced with a call to add points to the associated next block. At the end of the recursive method, each next block is recursed upon if not empty.

3.3.3 Handling intermediary methods

There can be an arbitrary number of intermediary methods from the enclosing loop to the recursive method. The original and transformed code for a intermediary method *foo* is shown in Figure 11. The method may have arbitrary prologues and epilogues, which must be transformed accordingly. For intermediary methods, both the prologue and epilogue are executed once per block in the transformed code, instead of once per point as in the original code. Therefore both the prologue and epilogue must be moved to a new loop (lines 2–6, 10–16 of Figure 11(b)). The new loops have access to the loop variant arguments that were added to the block. A graphical depiction of this change is shown in Figure 12 for a block size of 3. The prologue and epilogue are shown in white circles, and the next method call is shown in

```
1 Object foo(Point p, Object o1, Object o2, Node root) {
2   // do something - prologue
3   recurse(p, o1, o2, root);
4   // do something - epilogue
5   return ret;
6 }
```

(a) **Original code**

```
1  void foo(Object o1, Node root, Block b) {
2    for (int i = 0; i < b.size; i++) {
3      Point p = b.p[i];
4      Object o2 = b.o2[i];
5      // do something - prologue
6    }
7    BlockStack stack = /* stack instance */
8    stack.set[0].block = b;
9    recurse(o1, root, stack, 0);
10     for (int i = 0; i < b.size; i++) {
11     Point p = b.p[i];
12     Object o2 = b.o2[i];
13     // do something - epilogue
14     b.ret_foo[i] = ret;
15     continue;
16   }
17 }
```

(b) **Transformed code**

Figure 11. Intermediary methods

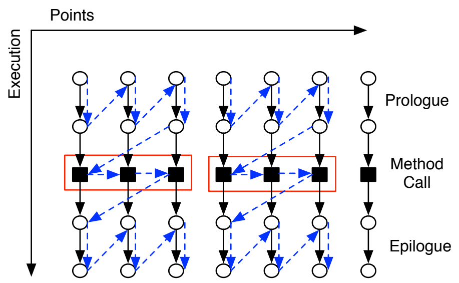


Figure 12. Iteration space prologues and epilogues

black squares. The original execution order is in full arrows, and the transformed execution order is in dashed arrows. The prologue is executed for all points in the block, then the transformed method is called once on the block shown as a rectangle. When the transformed method returns, the epilogue is executed for all points in the block.

Return statements within intermediary methods must be handled with care. To start, intermediary methods may no longer have return values, as they will be different per point. The return type of all intermediary methods are changed to void, and space to save the return value is reserved within the block. Each return statement must then be changed to an assignment of the return value, to the space reserved in the block (line 14 of Figure 11(b)). Then control flow must ensure that code after the return statement is no longer executed for that point. This is easier for epilogues. We can simply insert a continue statement, which will make the loop move on to the next point. (line 15 of Figure 11(b)).

Returns from method prologues require more careful changes. We need to invalidate the current point, which is already in the block, to prevent the next intermediary method and the epilogue from executing code for that point. We use a "valid" array within the block and mark prologue returns as invalid in that array. Subsequent intermediary methods and epilogues will skip invalid points. Finally, before entry to the recursive method, the block is compacted (all valid points are moved forward in the array so there are no invalid holes), to avoid the overhead of skipping over invalid points for the computation intensive portion of execution. This is not shown in Figure 11 for simplicity.

The block stack must be initialized before entry to the recursive method. This was done in line 11 of Figure 9 for no intermediary methods. When there are intermediary methods, the method just before the recursive method must initialize the block stack, as shown in line 8 of Figure 11(b).

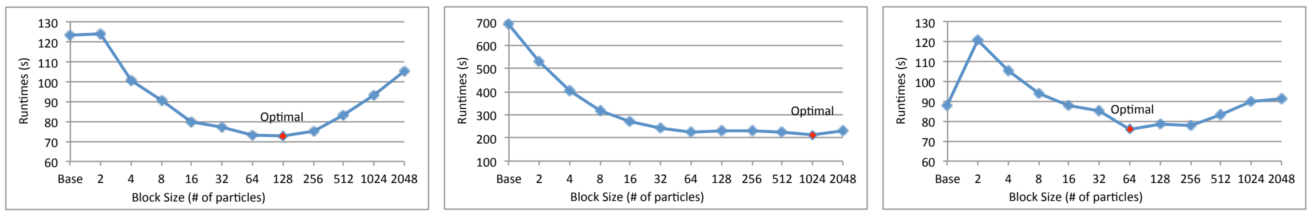
3.4 Discussion: generalization to DAGs and graphs

We note that TreeTiler identifies data structure traversals by looking for recursive traversals of recursive structures. While the discussion so far has focused on TreeTiler's application to traversals of trees, the framework's identification strategy may actually flag traversals of DAGs and general graphs as potential optimization targets, as well. Interestingly, the transformation presented here can be applied directly to these more general data structures; point blocking, if legal, will have similar locality effects regardless of the type of structure being traversed.

From the perspective of traversal algorithms, the key distinction between trees and the more general structures is that, in the latter case, depth-first traversals may visit the same node more than once. This means the correctness criteria for point blocking are more complex than for trees. Nevertheless, TreeTiler's sufficient condition of looking for parallelizable loops means that there will be no inter-traversal dependences, so point blocking will be correctly applied. We leave the problem of checking more complex correctness conditions, as well as an investigation of the efficacy of point blocking for DAG and graph traversals to future work.

4. Autotuning

In the previous section, we have presented an approach to identify recursive structures in tree traversal algorithms that can be transformed correctly, and a systematic method to transform the algorithm to enhance locality. Critical to the performance of the transformation is the block size B. The optimal block size is dependent on both machine parameters (*e.g.*, L1, L2 cache size) and algorithmic characteristics (*e.g.*, density of a block at different depths, which we call the *effective* block size). The notion of optimization parameters affecting the performance of transformations is well known. In recent years, there has been a large amount of research on *autotuning*, where a compiler automatically selects the best optimization parameters for a particular scenario [31, 33, 37]. In this section, we describe autotuning methods



(a) **Barnes-Hut** (b) **Point Correlation** (c) **Raytracing**

Figure 13. Runtime with varying block sizes for Barnes-Hut, Point Correlation, Raytracing on Opteron

that automatically select a good block size to use for the transformed algorithm.

Many autotuners can operate at compile time. For example, in dense linear algebra, tile size is dependent on machine parameters but independent of input characteristics and hence can be determined when the library is compiled [37]. Because the optimal block size for point blocking is dependent on input characteristics as well as machine parameters, TreeTiler's autotuners must operate at run-time, when the input is available.

4.1 Performance of various block sizes

In order to autotune different block sizes to choose the best size, we must have some idea of the behavior of different block sizes. Figure 13 shows the serial runtimes in seconds with varying block sizes for three benchmarks on an Opteron system with 128K of L1 cache and 1M of L2 cache. Intuitively, we would expect that a block size that is too small would perform poorly due both to the additional instruction overhead incurred by point blocking and to the fact that misses in the tree are incurred for every block (as discussed in Section 2)—more blocks will result in more misses in the traversal. However, if the block becomes too large to fit in cache, then we will begin to incur misses on the points instead. We thus expect there to be a "sweet spot," where the blocks are large enough to avoid most misses in the tree, but small enough to fit in cache, an expectation borne out by the results. In each figure, the best block size is highlighted, and is surrounded by block sizes that perform worse. The leftmost point on the x axis is the baseline, which corresponds to a block size of 1. For Raytracing, the baseline is faster than block sizes of 2 or 4 because it executes fewer instructions. For Barnes-Hut, enhanced locality almost counters instruction overhead for even a block size of 2, and for Point Correlation, the enhanced locality is decisive. This difference in behavior in benchmarks will be discussed in more detail in Section 5.

The valley shape of the graphs in Figure 13 suggests a hill climbing approach for finding the optimal block size. We can visualize the autotuner as descending a valley, profiling performance at different block sizes until the next largest block size degrades performance. The optimal block size is the lowest point in the valley. While this approach is attractive, we found that there is a subtle tradeoff between obtaining a representative sample of the points, and maintaining the locality of consecutive points, which militates against the hill climbing approach. This is discussed in the next section.

4.2 Random sampling

The optimal block size is dependent on not only machine parameters, but also algorithmic characteristics, which are often input dependent. The optimal block size for Barnes-Hut on one input may very well not be the optimal block size for Barnes-Hut on another input. Hence the autotuner must make decisions on a per input basis at runtime. The autotuner should consume only a fraction of the total number of points, or else the overhead of the autotuning phase could become significant. Complicating matters, the irregular nature of the algorithms means that different regions of the data structure might exhibit widely differing characteristics (*e.g.*, the traversals of points that walk one part of a tree might be much shorter than those that walk a different part of the tree). An autotuner that investigates various block sizes only using points from early in the execution may not see the full range of possible behaviors. The problem then, is to make a good decision representative of the all the points, while looking at only a fraction.

A common approach to account for this variability is to use random sampling. Randomly selecting the test points from among all the points provides on average the best representative of the entire set attainable. Because the points have been sorted so that consecutive points have similar traversals and enjoy temporal locality, we would like to take random samples of *blocks*. When testing a block size of 5 from 1000 points, we want to choose a random sequence of 5 consecutive points, rather than constructing the block from 5 randomly sampled points.

However, even this consideration is not enough. The next random block of 5 points sampled from the iteration space may exhibit little inter-block temporal locality. In contrast, in the actual execution, consecutively executed blocks will exhibit significant inter-block temporal locality, as their points are likely to have similar traversals. Thus, by introducing random sampling to account for input irregularity and make the autotuner's profiling more representative of actual execution, we may experience less locality, making the behavior *less* representative. We therefore investigate the performance of *two* autotuners, described in the next section, that trade off increased randomness of sampling and increased inter-block locality.

4.3 Implementation of autotuning

This section discusses the implementation of the autotuner in TreeTiler. As the transformations discussed in Section 3 are not useful without a good block size, the autotuner also needs to be integrated into TreeTiler. TreeTiler will generate transformed code for the tree traversal algorithm, and insert autotuning code before the enclosing loop (*e.g.* point loop) so that the best block size can be determined to be used for the point loop.

There should be a limit on the number of points used for autotuning to keep its overhead from becoming too high. We set the limit to maximum 1% of the total points. If the total number of points is too few, there might not be enough points for autotuning. We can apply a runtime check on the total number of points to decide whether to execute the autotuned and transformed code path at all. All the benchmarks we evaluate have one million points, of which 1% is $10{,}000$. We run each block size 5 times to average out irregularities, and this allows us to test up to a block size of 512. We also test the base case (the original code path), to check if we should be applying our transformation at all.

Hill climbing lets us use fewer points for autotuning if we arrive at an optimal block size before exhausting all the points allotted for tuning. However hill climbing requires us to test each block size 5 times consecutively before we know whether to test the next block size. Without random sampling, this method is susceptible to irregularities across the input. One way to distribute the irregularities across different block sizes is by consuming all 1% of the points, and testing block sizes in *interleaved* order. Due to the tradeoff discussed in Section 4.2 it is not clear whether random sampling should be used. Hence we implement two autotuners and compare their performance.

- **Auto-rand** uses random sampling with hill climbing
- **Auto-seq** uses sequential sampling with interleaved order

For **Auto-rand**, we use a hill climbing approach with a threshold of 20%. The autotuner starts at a block size of 8, and doubles the block size until the next block size takes 20% longer than the minimum recorded runtime, or the autotuner has consumed 1% of the total points. Then the best block size is compared with the base case. For benchmarks with a small optimal block size, the hill climbing approach can save points by consuming less than 1% of the total. The threshold was employed to ensure that noise in the profiling does not cause us to stop searching too early.

For **Auto-seq**, we always consume 1% of the total points, knowing in advance the maximum block size to test. Then the block sizes are interleaved starting with a block size of 8 to the maximum block size, to distribute input irregularities among the different block sizes. For example with a maximum block size of 32, the order of block size tests will be $8, 16, 32, 1, 8, 16, 32, 1$ ... (a block size of 1 is the base case), whereas for a hill climbing approach it would be $8, 8, 8, 8, 8, 16, 16, 16, 16, 16$ ... This results in more representative profiling at the cost of always consuming 1% of the total points for autotuning.

An important correctness condition for both **Auto-seq** and **Auto-rand** is that sampled points must be skipped in the point loop after the autotuning phase. This is because processing a point can have side-effects, and processing a point twice can be incorrect. Skipping points for **Auto-seq** is trivial. If P points have been used for autotuning, the point loop can simply start from the $P + 1$st point. Skipping points for **Auto-rand** is more complex because sampled points could be anywhere. When using **Auto-rand**, the transformed code skips over sampled points using the valid array mechanism described in Section 3.3.3.

Simply recording the time to process a test block does not consider the actual amount of work done (*e.g.* the traversal size) of the points in that block. The points in a block of 10 may traverse 100 nodes (on a per point basis) and take 10 ms, while the points in a block of 20 traverses 1000 nodes and takes 40 ms. In this case the latter is a better block size, and the recorded times should be normalized to the actual work done for a fair comparison. The actual work done is profiled in the recursive method as in line 28 of Figure 9. Profiling is done *only* for the autotuning phase to minimize overhead and prevent false sharing when the point loop is parallelized.

The autotuning loop is inserted at line 5 of Figure 9, and is explained in more detail in Appendix B.

5. Evaluation

Using the methodology described in the previous sections, TreeTiler is implemented as a Java source to source transformation. It takes a set of Java files as input, recognizes the recursive structure, performs the transformation, and outputs transformed Java files. TreeTiler can be configured to output transformed code with a fixed block size passed as an argument, or both flavors of autotuning described in Section 4. We implement TreeTiler using the JastAdd Extensible Java Compiler [7].

5.1 Evaluation Methodology

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

- **Base** is the baseline described for each benchmark below.
- **Block** is a TreeTiler output without autotuning, using an empirically determined optimal block size.

- **Auto-seq** is a TreeTiler output with autotuning using sequential sampling with interleaved test order.
- **Auto-rand** is a TreeTiler output with autotuning using random sampling with hill climbing test order.

Note that our baselines use standard optimizations proposed for enhancing temporal locality among consecutive points in tree traversal codes, as in [2, 29]. While these optimizations have been discussed for Barnes-Hut, we have applied analogous transformations to other benchmarks. Barnes-Hut uses a Hilbert space filling curve as in [2], and Point Correlation and Nearest Neighbor sorts the points in tree order as in [29]. Raytracing and Lightcuts schedule rays in chunks of 8×8 squares to enhance temporal locality among consecutive rays.

Barnes-Hut (BH) The Barnes-Hut algorithm is a scientific kernel for performing N-body simulation [3], and has been explained in detail in Section 2.1. We use the implementation from the Lonestar benchmark suite [17], augmented with the optimizations from [2], and the class C input, which has one million points.

Point correlation (PC) The two-point correlation is a spatial statistic that is of fundamental importance in many natural sciences. It is defined as the number of pairs of points in a dataset that lie within a given radius r of each other [12]. Finding the two-point correlation of a point can be accelerated by building a kdtree over the points, and pruning nodes when the minimum distance to the hyper-rectangle surrounding the node is larger than r [12]. Thus, PC involves repeated traversals of a kdtree. One million points are randomly generated in a three-dimensional space, and r is chosen so that the average correlation is 3732, or 0.37% of the total number of points. The benchmark finds the two-point correlation for all the points.

Nearest neighbor (NN) Nearest neighbor search is an optimization problem that arises often in data-mining, and involves finding closest points in metric spaces. NN is also accelerated by a kdtree, by pruning nodes that cannot be closer than the current closest find [12]. We implemented a NN kernel, using exclusion based pruning and the kdtree as discussed for PC. We randomly generate one million points in an 7-dimensional space, and find the nearest neighbor for all the points.

Raytracing (RT) Raytracing is a technique for rendering a scene by tracing the path of light through pixels in an image plane, and simulating the effects of the ray's encounters with scene objects. This can be accelerated using bounding volume hierarchies (BVHs), tree structures that permit fast determination of the objects a ray intersects. Our baseline is an optimized BVH-based raytracer from [35]. A random scene is generated with one million triangles. We rendered a screen of 1024×1024, which amounts to roughly one million rays.

Lighcuts (LC) Lightcuts is a scalable framework for computing realistic illumination when there are many light sources [36]. It uses the intuition in BH of approximating multiple gravitational objects as one, by approximating multiple light sources as one. A binary tree of lights (*e.g.* light tree) is constructed, and locally adaptive light cluster partitions are computed per ray by traversing the light tree, while adhering to a fixed error bound. We used the Lightcuts implementation from [30]. While the original Lightcuts paper [36] renders scenes with up to $600,000$ lights, we found the implementation we obtained takes a very long time to render just $1,000$ lights. With only $1,000$ lights, the average traversal isn't deep enough for TreeTiler to be effective. We use Lightcuts with a screen of 1024×1024 and 16 lights to demonstrate that TreeTiler can autotune and choose the base case, where the transformation is not applied. The Lightcuts source code consists of 62 files, and is the most complex of our benchmarks. Nevertheless, TreeTiler can correctly transform the relatively complex code of Lightcuts.

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

- 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 64 threads, at which point our system is employing 4-way multithreading.
- The **Opteron** system runs Linux 2.6.24 and contains four 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 8 threads.

TreeTiler is independent of any parallelization model, and takes as input sequential code, and outputs sequential code. As discussed in Section 3.2, parallelizability is a *sufficient* condition for point blocking to be legal, not a necessary one. Because we want to test our benchmarks on multicores, we have manually parallelized both the benchmark baselines and the three TreeTiler variants using the *foreach* construct of the Galois system [18]. This *foreach* construct is implemented internally by Java threads, and is analogous to an OpenMP for loop. We simply add this *foreach* construct to the point loop, and it parallelizes by processing multiple points or blocks at once. The autotuning phase is not parallelized. We apply load balancing with work stealing implemented via lock-free double-ended queues as in Cilk [9]. The granularity of work chunks is equal to the empirically determined optimal block size of **Block** for **Base** and **Block**, and the autotuned block size for **Auto-seq** and **Auto-rand**. The benchmarks were written in Java 6 and executed on the Java HotSpot VM version 1.6. A 12GB heap was used. To account for the effects of JIT compilation, each configuration was run 10 times, and the average of the latter 7 runs was recorded. We show standard deviations of our tests in Appendix C to support their statistical reliability. For each

Version	Cycles (millions)	Instructions (millions)	CPI	L1D miss rate(%)	L2 miss rate(%)
Base	1360984	782890	1.74	8.38	55.30
Block	498812	667840	0.75	1.32	18.14

Table 2. Performance counters for BH

benchmark, only the traversal phases were timed, and a full GC (garbage collection) was forced before timing to minimize the effects of GC in the autotuning phase. GC time is excluded in the reported times.

5.2 Experimental Results

5.2.1 Barnes-Hut

Figure 14 shows the results for BH. Figure 14(a) shows speedups of the transformed versions compared to the serial baseline on the Niagara system. Figure 14(b) shows % improvement of the transformed versions compared to the parallel baseline on the Niagara system. Figure 14(c) shows speedups of the transformed versions compared to the serial baseline, and Figure 14(d) shows % improvement of the transformed versions compared to the parallel baseline on the Opteron system. This order of Figures will be used for all the benchmarks.

Block performs best with % improvement of 76.4% and 76.9% on the Niagara and Opteron respectively. The autotuners are slightly worse than the empirically determined block size as expected, and **Auto-seq** with 70.8% and 77.9% is slightly better than **Auto-rand** with 61.9% and 62.8% respectively on the Niagara and Opteron.

These improvements are sustained as we increase the number of threads for the Opteron system. For the Niagara system, we note that the transformed versions' advantage over the baseline tapers off as the number of threads increases beyond 16 threads. This is because on 32–64 threads, the Niagara uses 2–4-way multithreading. The Niagara's implementation of multithreading is meant to hide latency: when one thread stalls due to a cache miss, the second thread can execute. As the Niagara is already hiding latency through multithreading, it obviates the need for our transformations, which hide latency through restructuring.

To verify that the improvement of TreeTiler is indeed from enhanced locality, we used Intel's VTune profiling framework to access the performance counters on the Pentium system used in Table 1. The empirically determined optimal block size for this system was 64. The performance counter results are shown in Table 2. They show a drastic reduction in CPI and cache misses with a performance improvement of 172%, which is even larger than reported for the Niagara and Opteron. We also note a reduction in instructions, due to fewer accesses of the tree. While TreeTiler incurs overhead by adding instructions to access points within a block, it can save instructions in accessing nodes of the tree when points of a block have similar traversals.

5.2.2 Point correlation

Figure 15 shows the results for PC. We attain improvements of up to 182.3%, 237.2% and 244.8% for **Block**, **Auto-seq** and **Auto-rand** respectively on the Niagara. The maximum improvements are 211.1%, 234.7% and 211.7% for **Block**, **Auto-seq** and **Auto-rand** respectively on the Opteron. On the Niagara we start with modest improvements for 1–4 threads, but the improvement is greatly enhanced for more than 8 threads. We speculate that this is due to bus saturation. PC performs fewer instructions per point per node compared to BH, and hence is expected to be more bandwidth-hungry. This demonstrates another advantage of our implementation: because the tiling transformation reduces cache misses, it reduces bus pressure. Hence, the optimized implementation can perform more operations before saturating the bus. The Niagara is designed with more bus bandwidth than the Opteron. Hence we see a saturation of the bus at 8 threads for the Niagara whereas, we see 200% improvement due to bus saturation on the Opteron right away starting from 1 thread. The trend of diminishment in improvement for 32–64 threads where the Niagara is hiding cache miss latency through multithreading is evident here as in BH.

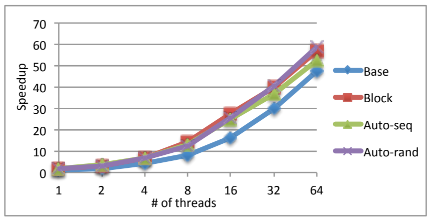
On the Niagara at 16–32 threads, the autotuners perform better than the empirically determined optimal block size. This suggests that the optimal block size is dependent on the number of threads, which we do not factor in currently. Currently the autotuning phase is performed sequentially. We would expect the number of threads to affect the optimal block size on systems like the Niagara that have shared L2 caches. In this case, the autotuners were "lucky," and chose a better block size for 16–32 threads.

5.2.3 Nearest neighbor

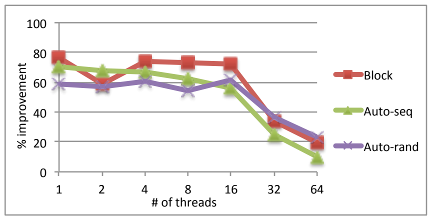
Figure 16 shows the results for NN. We attain improvements of up to 84.7%, 69.4% and 76.5% for **Block**, **Auto-seq** and **Auto-rand** respectively on the Niagara. The maximum improvements are 117.2%, 96.9% and 97.9% for **Block**, **Auto-seq** and **Auto-rand** respectively on the Opteron. On the Niagara, we see the trends of bus saturation at above 8 threads giving more improvement for the transformed versions, and multithreading decreasing the improvements at 32–64 threads. The autotuners perform significantly worse than **Block** for 64 threads. This is because the autotuners consume 1% of the total points for autotuning, and these points are processed sequentially. From Amdahl's law, the speedup attained with 99% of the work fully parallelized among 64 processors is 39.2, and this sequential autotuning phase limits the performance of the autotuners at many threads.

5.2.4 Raytracing

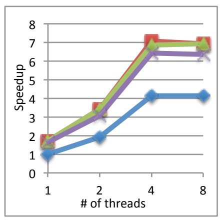
Figure 17 shows the results for RT. The maximum improvements are small, 3.3%, 0.2% and −1.2% for **Block**, **Auto-seq** and **Auto-rand** respectively on the Niagara, and 17.5%,



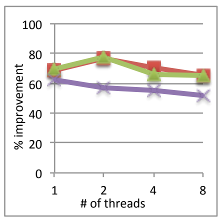
(a) **Speedup vs serial on Niagara**



(b) **% improvement vs parallel on Niagara**

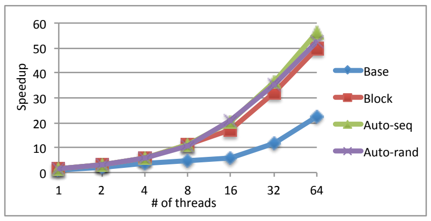


(c) **Speedup vs serial on Opteron**

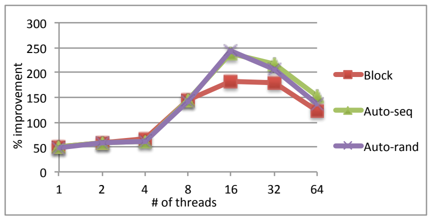


(d) **% improvement vs parallel on Opteron**

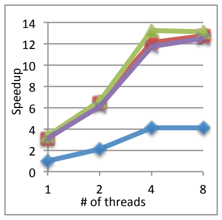
Figure 14. Results for Barnes-Hut



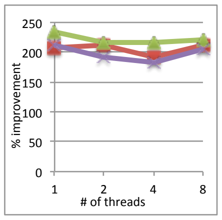
(a) **Speedup vs serial on Niagara**



(b) **% improvement vs parallel on Niagara**

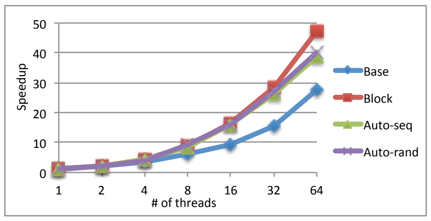


(c) **Speedup vs serial on Opteron**

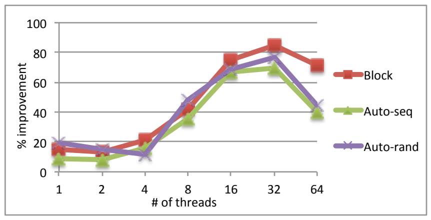


(d) **% improvement vs parallel on Opteron**

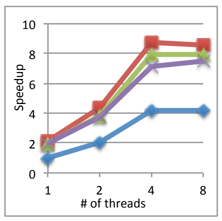
Figure 15. Results for Point Correlation



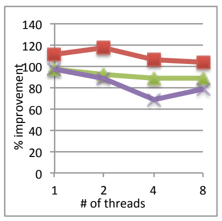
(a) **Speedup vs serial on Niagara**



(b) **% improvement vs parallel on Niagara**

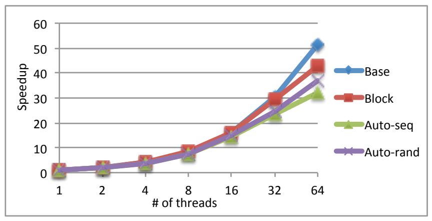


(c) **Speedup vs serial on Opteron**

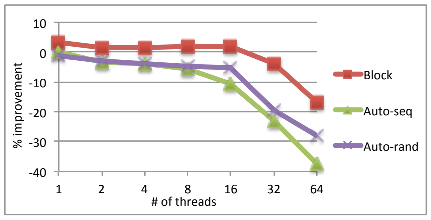


(d) **% improvement vs parallel on Opteron**

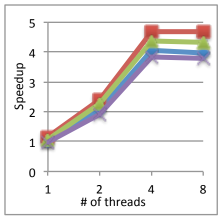
Figure 16. Results for Nearest Neighbor



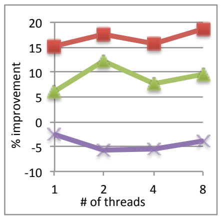
(a) **Speedup vs serial on Niagara**



(b) **% improvement vs parallel on Niagara**

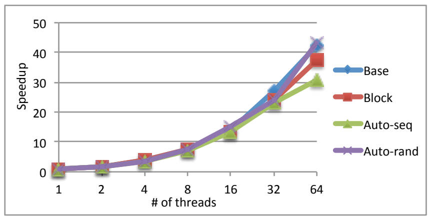


(c) **Speedup vs serial on Opteron**

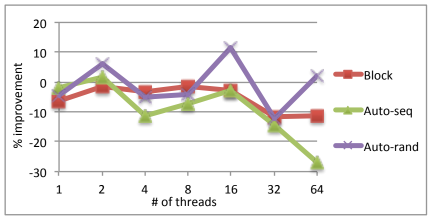


(d) **% improvement vs parallel on Opteron**

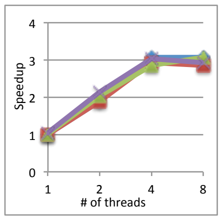
Figure 17. Results for Raytracing



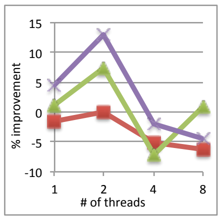
(a) **Speedup vs serial on Niagara**



(b) **% improvement vs parallel on Niagara**



(c) **Speedup vs serial on Opteron**



(d) **% improvement vs parallel on Opteron**

Figure 18. Results for Lightcuts

Benchmark	# Files	LOC	Transform time (ms)	Total time (ms)
BH	5	364	8.3	998
PC	5	390	6.2	975
NN	3	367	5.4	810
RT	38	3810	10.8	1798
LC	59	4291	11.2	2342

Table 3. Lines of code and transformation times

12.7% and 2% for **Block**, **Auto-seq** and **Auto-rand** respectively on the Opteron. For the Niagara, the autotuners choose the base case sometimes. Because both the empirically determined block size and the autotuners only consider sequential performance, the selected block size can yield performance degradations when used in parallel. This can be mitigated by parallelizing the autotuners, which we leave for future work. We discuss why RT shows less improvement compared to the previous benchmarks in Section 5.3

5.2.5 Lighcuts

Figure 18 shows the results for LC. We noted previously that LC has a very small average traversal, and we do not expect our transformation to be effective. For small average traversals, the choice of block size is a secondary effect, and we chose 32 for both systems. LC has a higher deviation in runtimes than RT due to more computation per node, which includes caching rays for fast radiance computation. Hence the 10% pluses and minuses in improvement can be considered within the noise range, and all versions perform more or less similarly. The autotuners often choose the base case, and, with the exception of 64 threads on the Niagara where Amdahl's law hurts the autotuners, the autotuners do not perform worse than the baseline.

5.2.6 Transformation times

Table 3 shows the lines of code and transformation times for our benchmarks. We show both the time it takes to perform our transformation phase, and the total time including file I/O and parsing. Our transformation phase has very small overhead, amounting to less than 1% of the parsing time.

5.2.7 Block sizes

Table 4 shows the empirically determined optimal block size, and the average block size chosen by each autotuner. The autotuner block sizes are the average of the block sizes chosen for the recorded latter 7 of 10 runs. The top 7 rows are the Niagara, and the bottom 4 rows are the Opteron. The results generally show that **Auto-seq** chooses a block size closer to the optimal block size than **Auto-rand**, implying that in the tradeoff discussed in Section 4.2, it is better to attend to locality among samples at the start of the point set, than obtaining randomized samples from the entire point set. Although not decisive, this is also the general trend of the experimental results in Figures 14–18. For RT on the Niagara, and LC on both systems, **Auto-rand** reverts to the base case successfully on many occasions. **Auto-rand** is more successful in reverting to the base case because it can better handle irregularities in the initial points with random sampling, and *always* reverts to the base case for LC on the Niagara.

Benchmark	# Objects	Tree type	Paths	Traversal size	
				# Nodes	Bytes
BH	1000000	OctTree	Many	2708.78	139616
PC	1000000	KdTree	Many	4070.77	183422
NN	1000000	KdTree	Few	1950.55	218461
RT	1000000	BVH	Few	909.96	21839
LC	16	BinaryTree	Many	17.27	1796

Table 5. Average traversal sizes for each benchmark

5.3 Traversal and effective block sizes

We have shown that TreeTiler attains impressive improvements for some benchmarks, while for others, even an empirically determined optimal block size is no better than the baseline. In Table 1, we saw that the L2 miss rate of an untransformed code increases as the average traversal gets larger. In this section we examine the average traversals of each of our benchmarks, and introduce *effective block size* as a metric to gauge the similarity of consecutive traversals.

Table 5 shows the average traversal sizes for each of our benchmarks. The objects are the entities used to create the data structures being traversed. We show traversal size in both the number of data structure nodes traversed, and the number of bytes accessed within the nodes. We note that BH and PC have larger traversals (both in # nodes and bytes) because processing a single point requires deeply traversing many paths through the data structure. On the other hand NN and RT have smaller traversals because the algorithms are essentially guided searches, and processing a single point essentially takes a single path through the data structure. NN has a larger memory footprint relative to its # nodes because the dimensionality is 7. While we expect LC to have similar behavior to BH and PC, we were not able to test enough objects due to performance limitations.

As discussed in Section 2.2, large traversal sizes result in disastrous LRU cache replacement when the average traversal does not fit in cache. TreeTiler reduces node misses by processing multiple points at once, reducing the number of node accesses. Benchmarks with large traversal sizes are where we expect TreeTiler to yield the most improvement, and this is borne out by our results. PC, NN and BH attain improvements of up to 245%, 117% and 76% whereas there is not much improvement for RT and LC. We see *degradation* in performance for RT and LC at 32+ threads because there is not have much headroom for optimization, and because the autotuning phase is done serially. This serial phase significantly limits speedup according to Amdahl's Law.

When the average traversal is too large to fit in cache, TreeTiler tries to fit a block of points in cache instead. The actual block size will vary as the traversal progresses, and the points within the block diverge on different paths. Therefore what TreeTiler should target to fit in cache is not the initial block size, but the average block size across the en-

Threads	BH			PC			NN			RT			LC		
	Opt	A-seq	A-rand	Opt	A-seq	A-rand	Opt	A-seq	A-rand	Opt	A-seq	A-rand	Opt	A-seq	A-rand
1	32	27	11	32	32	82	4096	512	512	16	32	2	1	16	1
2	32	23	11	32	46	64	4096	512	440	16	32	3	1	16	1
4	32	30	18	32	32	64	4096	512	296	16	32	11	1	16	1
8	32	30	11	32	73	64	4096	512	512	16	32	3	1	16	1
16	32	18	18	32	37	64	4096	512	440	16	32	3	1	16	1
32	32	32	18	32	41	64	4096	512	440	16	32	15	1	16	1
64	32	32	15	32	37	73	4096	512	440	16	30	11	1	16	1
1	128	82	238	1024	512	512	2048	512	512	64	128	111	1	8	2
2	128	73	207	1024	457	512	2048	512	475	64	128	102	1	91	19
4	128	73	113	1024	512	403	2048	512	403	64	119	183	1	16	2
8	128	91	184	1024	457	512	2048	512	475	64	128	210	1	8	1

Table 4. Block sizes for **Block**, **Auto-seq** and **Auto-rand**

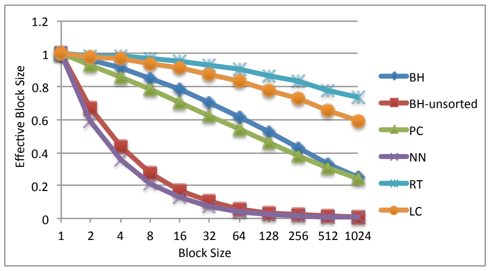


Figure 19. Effective block sizes for each benchmark

tire traversal. We measured *effective block size*, the average of the actual block size at the beginning of each recursive method call. Figure 19 shows the *normalized effective block size* for 6 benchmarks across a range of initial block sizes, where *normalized effective block size = effective block size / initial block size*. Normalized effective block size is roughly a measure of the convergence of traversals. The maximum value of 1 means that traversals of all points with a block are identical. Normalized effective block size gets smaller as the initial block size gets larger, because more initial points allow more divergence among them.

We have shown the normalized effective block size for BH with and without sorting to demonstrate the importance of sorting optimizations [2]. Not sorting for BH results in a much smaller normalized effective block size, and hence much less similarity in consecutive traversals. As a result, while the optimal initial block size is 128 when transforming our BH baseline, the optimal block size is 2048 when transforming an unoptimized baseline. The effective block sizes corresponding for each are 68.53 and 8.92, where sorted BH has a larger effective block size with a smaller initial block size. The relation is not linear due to various irregular effects, but do suggest some trends. The normalized effective block size is very small for NN because at each recursive call the block is potentially split into two blocks due to different orders of traversing children illustrated in Figure 10. A very small normalized effective block size implies that the optimal initial block sizes should be very large, and they are found to be up to 4096 for NN.

6. Related Work

6.1 Locality Transformations

Salmon used Orthogonal Recursive Bisection [8] to directly partition the point space to provide physical locality [28]. Singh *et al.* recognized that N-body problems already have a representation of the spatial distribution encoded in the tree data structure and partitioned the tree instead of partitioning the point space directly [29]. Amor *et al.* exploited locality among points by linearizing them using space filling curves [2]. Han *et al.* proposed computation reordering in Z-curve order (Z-SORT) that has better performance than lexical sort at the cost of more overhead [14]. All of these approaches improve locality up to a point, as discussed in Section 2.1, and both our baseline and transformed code exploit some form of point sorting.

Singh *et al.* also proposed *costzones* to improve load balance across multiple Barnes-Hut timesteps; we expect their effects are largely orthogonal to the transformations presented here. Amor *et al.* proposed communication optimizations for distributed memory systems. While we evaluate our techniques on shared memory systems, we expect similar improvements if applied to an optimized distributed memory implementation.

Pharr *et al.* addressed locality issues in raytracing where the scene is too large to fit in memory [24]. They proposed caching and lazy creation of texture and geometry, and grouping rays into groups ("voxels") to account for spatial coherence between rays. Rays are partially traced on a per voxel basis, and voxels are scheduled to maximize locality in already created texture and geometry. Mansson *et al.* examined various heuristics for grouping *secondary* rays, which are reflected from primary rays, to enhance locality in deep raytracing [22]. Their heuristics are in essence a way of *sorting* secondary rays. These approaches are very application specific, neither are general transformations like ours.

Various techniques have been proposed to enhance *spatial* locality in dynamic data structures [4–6, 14, 20, 23, 32]. Chilimbi *et al.* proposed techniques for using programmer annotations to allocate subtrees to cache lines [5], and moving objects and fields around at GC time for spatial locality [6]. Lattner *et al.* proposed a technique that uses a context-sensitive pointer analysis to segregate distinct instances of heap allocations into separate memory pools, which improves *spatial* locality for programs which allocate multiple pointer based data structures but traverse only one at a time [20]. While their work is of less utility for most benchmarks discussed in this paper (with the exception of LC), which allocate only a single tree, it is orthogonal to our work, and we expect it to be fruitful for more complex applications with multiple pointer based data structures. Mellor-Crummey *et al.* proposed a combination of data reordering and computation reordering to improve memory hierarchy performance for n^2 interaction algorithms [23]. Han *et al.* proposed a data reordering algorithm, GPART, that applies hierarchical clustering on data without geometric coordinate information [14]. We expect techniques to enhance spatial locality to have positive effects on both our baseline and transformed code, if the cost of data reordering can be amortized over the computation.

6.2 Vectorization Transformations

Hernquist vectorized Barnes-Hut across nodes of the tree, so that each point traverses all nodes at the same level simultaneously [15]. This approach effectively changes the order of the tree traversal from depth-first to breadth-first. This has two drawbacks. First, it changes the traversal order of the tree, affecting the result in the presence of non-commutative operations (such as floating-point addition). Second, there typically are not many nodes per tree level, leading to short vectors (and less parallelism).

Makino vectorized the tree traversal across points, instead, leading to a per-point parallelization similar to our baseline [21]. An interesting aspect of Makino's approach is that to enable vectorization, the code is transformed in a manner similar to the loop interchanged implementation described in Section 2.2. However, there are a few key points to note. First a simple loop interchange does not suffice to exploit locality. Second, Makino's transformation relies on a pre-computed traversal of the tree, and changes the order in which particular tree nodes are visited by different points, reducing the generality of his transformation.

Work on vectorizing Barnes-Hut have naturally extended to GPU implementations of n-body algorithms [13, 19]. These implementations generally group many points in the leaves of the tree, so that the points within a single leaf are ensured to have identical interaction lists and can be divided among processing units. The interaction lists are computed on the CPU and sent to the GPU for mass parallel force computation. The GPU's natural execution model results in traversals of the interaction list similar to tiling. However, computing the interaction lists still requires traversing the tree, and the locality penalties of a naïve traversal remain.

6.3 Other Tree Traversal Transformations

Aluru *et al.* discussed changing the tree structure of Barnes-Hut to improve performance [1]. We note that our transformations are independent of the type of tree used (indeed, the tree in raytracing is different from that in Barnes-Hut), and hence our approach can apply to their algorithm as well.

Rinard and Diniz used a commutativity analysis to parallelize an N-body code in a unique manner [26]. Rather than distributing the points among threads, they are able to prove through compiler analysis that updates to the points commute, and hence multiple threads can update points simultaneously. This is akin to parallelizing the traversal loop in our abstract model, rather than the point loop.

Ghiya *et al.* proposed an algorithm to detect parallelism in C programs with recursive data structures [10]. These tests rely on shape analysis to provide information on whether the data structure is a tree, DAG or general graph, and apply different dependence tests depending on data structure shape. Their analyses focus on parallelization and do not consider locality, but we believe their approaches might inform an automatic transformation framework that implements our techniques.

7. Future work and conclusions

7.1 Future work

There is ample opportunity for further investigation in this area. It seems intuitive that the best optimization parameters for point blocking should be related to architectural parameters such as cache size and application attributes such as traversal sizes and effective block sizes; it may be possible to devise a simple analytical model based on a small number of measurable parameters that can short-circuit the "guess and check" autotuning process implemented by TreeTiler and arrive at an effective solution more quickly.

TreeTiler currently implements a sufficient, but not necessary, check for the legality of point blocking. Just as there is a rich set of conditions for the legality of various loop transformations in regular codes, there are a similar set of conditions for the legality of point blocking. An interesting avenue of future work is augmenting TreeTiler to evaluate these more complex conditions. This will likely require a shape analysis, and presents an interesting application for existing analyses.

In general, this work elucidates that, even in irregular applications, there may be significant structure that can be exploited to improve locality. Just as we found that an analog of loop tiling can provide substantial benefits to traversal codes, an open question is whether there are other similarly analogous transformations that can be applied to irregular applications in a general manner.

7.2 Conclusions

In this paper, we demonstrated that, despite their seeming irregularity, many traversal codes, which perform repeated traversals of data structures such as trees and graphs, possess a common algorithmic structure that admits substantial data reuse. We thus developed a novel optimization called *point blocking* that exploits this data reuse. Popular existing locality enhancing techniques for these codes lose their effectiveness as the traversal sizes increase. However, point blocking, much like its regular analog, loop tiling, is able to continue exploiting locality regardless of traversal size.

We developed an automatic transformation and optimization framework called TreeTiler that determines when point blocking can be applied and automatically transforms an application to leverage the technique. TreeTiler then uses run-time autotuning to select transformation parameters to best exploit locality. We showed that TreeTiler can successfully automatically transform and tune a set of five benchmark traversal codes, achieving performance improvements of up to 245% over optimized parallel baselines. Furthermore, these performance gains persist as the applications scale, and, in fact, applications transformed by TreeTiler can deliver better scalability than the untransformed baselines.

Acknowledgments

The authors would like to thank Bruce Walter for providing the Raytracing benchmark code, and Sebastian Thees for providing the Lightcuts benchmark code. We would also like to thank the anonymous referees for providing insightful comments. This research was supported in part with funding from Intel and a grant from the Purdue Research Foundation.

References

[1] S. Aluru, J. Gustafson, G. M. Prabhu, and F. E. Sevilgen. Distribution-independent hierarchical algorithms for the n-body problem. *J. Supercomput.*, 12:303–323, October 1998.

[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*, PARA '00, pages 342–349, London, UK, 2001. Springer-Verlag.

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

[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*, PLDI '99, pages 13–24, New York, NY, USA, 1999. ACM.

[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*, PLDI '99, pages 1–12, New York, NY, USA, 1999. ACM.

[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*, ISMM '98, pages 37–48, New York, NY, USA, 1998. ACM.

[7] 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*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[8] G. C. Fox. A graphical approach to load balancing and sparse matrix vector multiplication on the hypercube. *Institute for Mathematics and Its Applications*, 13:37–+, 1988.

[9] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998.

[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, New York, NY, USA, 1996. ACM.

[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)*. MIT Press, 2001.

[13] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji. 42 tflops hierarchical n-body simulations on gpus with applications in both astrophysics and turbulence. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[14] H. Han and C.-W. Tseng. Exploiting locality for irregular scientific codes. *IEEE Trans. Parallel Distrib. Syst.*, 17:606–618, July 2006.

[15] L. Hernquist. Vectorization of tree traversals. *J. Comput. Phys.*, 87:137–147, March 1990.

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

[17] 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.

[18] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *SIGPLAN Not. (Proceedings of PLDI 2007)*, 42(6):211–222, 2007.

[19] M. H. L. Nyland and J. Prins. Fast n-body simulation with cuda. *GPU Gems*, (3):677–695, 2007.

[20] 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*, PLDI '05, pages 129–142, New York, NY, USA, 2005. ACM.

[21] J. Makino. Vectorization of a treecode. *J. Comput. Phys.*, 87:148–160, March 1990.

[22] 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, Washington, DC, USA, 2007. IEEE Computer Society.

[23] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *Int. J. Parallel Program.*, 29(3):217–247, 2001.

[24] 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*, SIGGRAPH '97, pages 101–108, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[25] K. Pingali, M. Kulkarni, D. Nguyen, M. Burtscher, M. Mendez-Lojo, D. Prountzos, X. Sui, and Z. Zhong. Amorphous data-parallelism in irregular algorithms. Technical Report TR-09-05, Department of Computer Science, The University of Texas at Austin, February 2009.

[26] 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.

[27] 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.

[28] J. K. Salmon. *Parallel hierarchical N-body methods*. PhD thesis, Pasadena, CA, USA, 1991.

[29] 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.

[30] S. Thees and C. Weiland. Implementing lightcuts. Technical report, Fachhochschule Bonn-Rhein-Sieg, University of Applied Sciences, Fachbereich Informatik, July 2008.

[31] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[32] 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*, PACT '98, pages 322–, Washington, DC, USA, 1998. IEEE Computer Society.

[33] R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1), 2005.

[34] I. Wald. On fast construction of sah-based bounding volume hierarchies. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 33–40, Washington, DC, USA, 2007. IEEE Computer Society.

[35] 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.

[36] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D. Greenberg. Lightcuts: a scalable approach to illumination. *ACM Transactions on Graphics (SIGGRAPH)*, 24(3):1098–1107, July 2005.

[37] C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the atlas project. *Parallel Computing*, 27:2001, 2000.

A. Block class implementation

Figure 20 shows the implementation of the automatically generated block classes corresponding to Figure 9. The Block class allocates space for the loop variant arguments passed to the recursive method, as well as the return values of all intermediary methods. The BlockSet class has a reference to the current block, and allocates space for the next blocks of the next level. The BlockStack class is simply an array of BlockSets.

B. Autotuning code sample

Figure 21 shows the autotuning code inserted before the point loop of the transformed generic algorithm (at line 5 of Figure 9). With the exception of explicit code to skip over sampled points for **Auto-rand** (lines 30, 35), the code is common for both **Auto-seq** and **Auto-rand**. The autotuner is integrated into the *Block* class. The autotuning loop is a replica of the point loop with some additions. The point p is now drawn from a *samplePoint* set by the autotuner (line 5). Calls to *tuneEntry* and *tuneExit* are inserted before and after the call path to the recursive method (lines 10, 16), to record the time it took to process a test block. *tuneExit* configures the next block size, and additionally sets a random sample point for **Auto-rand**. *tuneExit* returns true if the autotuning is complete to break out of the autotuning loop. *nextSample* (line 25) increments *samplePoint*, and additionally skips over sampled points for **Auto-rand**. The original code path is used for testing the base case (line 12). If the base case is better than the best block size, the base case is used for the point loop (lines 28-32).

C. Deviations of experimental tests

Table 6 shows the % of the standard deviations of the runtimes to the average runtime. The statistics are recorded for the latter 7 of 10 runs. #T is the number of threads and Ba, Bl, As, Ar denotes **Base**, **Block**, **Auto-seq**, **Auto-rand** respectively. The top 7 rows are the Niagara, and the bottom 4 rows are the Opteron. **Auto-rand** has the largest deviation in general because it is random and dependent on the samples. Excluding the autotuners, the maximum deviation is 9.2% of the average runtime, and generally much smaller, implying that our results are statistically stable.

#T	BH				PC				NN				RT				LC			
	Ba	**Bl**	**As**	**Ar**	**Ba**	**Bl**	**As**	**Ar**	**Ba**	**Bl**	**As**	**Ar**	**Ba**	**Bl**	**As**	**Ar**	**Ba**	**Bl**	**As**	**Ar**
1	1.99	0.56	0.93	4.09	0.23	1.31	1.65	2.65	0.75	0.52	0.37	0.52	1.62	0.26	0.10	0.43	2.69	2.83	3.30	3.29
2	0.24	0.17	0.36	3.60	0.26	0.14	0.72	0.16	0.33	0.45	0.27	8.24	0.11	0.17	0.16	0.44	3.61	4.02	3.87	2.85
4	0.08	0.15	0.22	5.31	0.20	0.12	0.07	0.13	0.43	0.69	0.38	14.76	0.06	0.10	0.08	1.62	3.61	4.02	3.87	2.85
8	0.29	0.16	0.22	3.59	0.40	0.14	0.50	0.22	1.13	0.64	0.31	0.72	0.06	0.07	0.06	2.16	3.61	4.02	3.87	2.85
16	0.32	0.21	0.28	3.41	0.45	0.17	2.08	0.46	1.04	1.63	0.4	16.73	0.14	0.39	0.18	2.60	3.51	4.49	4.22	2.67
32	0.61	0.14	0.43	3.22	0.38	0.28	1.02	0.51	0.96	1.1	2.12	11.56	0.57	2.88	1.36	13.28	3.81	4.00	3.36	10.67
64	1.39	2.86	0.38	2.59	0.24	1.32	0.90	1.07	0.91	1.36	2.16	5.24	1.66	1.79	1.02	21.46	4.02	5.03	6.53	2.88
1	8.45	4.89	3.85	6.87	1.80	6.71	0.34	1.76	1.88	0.73	1.14	0.75	2.33	1.00	6.15	5.76	7.83	3.64	4.01	4.09
2	4.48	1.42	3.05	15.55	11.65	0.27	1.67	0.22	9.23	1.34	2.67	2.69	1.96	0.72	3.72	5.62	7.80	4.14	4.01	3.92
4	0.60	1.03	0.52	7.83	0.17	0.12	0.96	14.39	0.68	1.89	0.45	25.07	1.10	0.32	0.72	1.73	3.61	3.30	5.58	4.23
8	0.91	0.60	0.84	9.20	2.15	0.19	1.36	0.23	0.77	0.38	0.38	1.48	1.38	5.48	0.45	0.79	2.67	4.50	4.04	2.99

Table 6. Standard deviation of runtimes, as % of average runtime

```
class Block {
  int size;
  Point [] p;
  Object [] o2;
  Object [] ret_foo;

  void add(Point p_, Object o2_) {
    p[size] = p_;
    o2[size] = o2_;
    size++;
  }

  void recycle() {
    size = 0;
  }
}

class BlockSet {
  Block block;  // just reference
  Block nextBlock = /* actual allocation */
  // more nextBlocks if necessary
}

class BlockStack {
  BlockSet [] set;
}
```

Figure 20. Implementation of block classes

```
Set<Point> points = /* entities in algorithm */
Object o1 = /* something loop invariant */
Block b = /* block instance */
while (true) {
  Point p = points[Block.samplePoint];
  Object o2 = /* something loop variant */
   // do something - prologue
   b.add(p, o2);
   if (b.size == blockSize) {
    Block.tuneEntry();
    if (blockSize == 1) {
      foo(p, o1, o2, root);
    } else {
      foo(o1, root, b);
    }
    if (Block.tuneExit()) break;
    for (int i = 0; i < b.size; i++) {
      Point p = b.p[i];
      Object o2 = b.o2[i];
      Object o3 = b.ret_foo[i];
      // do something - epilogue
    }
    b.recycle();
  }
  Block.nextSample();
}
int i = Block.samplePoint;
if (blockSize == 1) {
  for ( ; i < points.length; i++) {
    if (Block.isSampled(i)) continue;
    // original code
  }
} else {
  for ( ; i < points.length; i++) {
    if (Block.isSampled(i)) continue;
    // transformed code
  }
}
```

Figure 21. Autotuning code sample