Locality Transformations for Nested Recursive Iteration Spaces

Kirshanthan Sundararajah   Laith Sakka   Milind Kulkarni

School of Electrical and Computer Engineering
Purdue University
{ksundar, lsakka, milind}@purdue.edu

Abstract

There has been a significant amount of effort invested in designing scheduling transformations such as loop tiling and loop fusion that rearrange the execution of dynamic instances of loop nests to place operations that access the same data close together temporally. In recent years, there has been interest in designing similar transformations that operate on recursive programs, but until now these transformations have only considered simple scenarios: multiple recursions to be fused, or a recursion nested inside a simple loop. This paper develops the first set of scheduling transformations for *nested recursions*: recursive methods that call other recursive methods. These are the recursive analog to nested loops. We present a transformation called *recursion twisting* that automatically improves locality at all levels of the memory hierarchy, and show that this transformation can yield substantial performance improvements across several benchmarks that exhibit nested recursion.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation, Compilers, Optimization

Keywords Nested Recursion; Locality Optimization

1. Introduction

Over the last several decades, there has been a substantial amount of effort invested in designing *locality enhancing* compiler transformations: transformations that can restructure code to improve locality of reference, and hence improve cache behavior [1]. Unlike transformations that change the operations of a piece of code, these locality-enhancing transformations are *scheduling* transformations: they rearrange the order of the existing operations in a piece of code without changing what the operations do. By rescheduling computations to ensure that operations that touch the same piece(s) of data occur close together in time, these transformations promote good cache behavior. For programs with substantial amounts of data reuse, these transformations are crucial for achieving good performance.

Common frameworks for performing rescheduling transformations include the unimodular framework [3] and the polyhedral framework [5, 12, 13]. These frameworks operate by conceptualizing the dynamic execution of a set of nested loops as an *iteration space*, with each point in that iteration space representing a *dynamic instance* of the loop body. Each of these dynamic instances represents the same static code, but executing on a piece of data determined by the loop indices. Transformations such as loop interchange, loop skewing, and loop tiling can be understood as transformations on this iteration space that change the order of iteration through this space. Crucially, all of these frameworks only handle *affine* iteration spaces: iteration spaces that arise from the use of nested loops.

For-loops are not the only source of dynamic instances of the same (static) operations, however. Recursive methods also result in the same code being repeatedly re-invoked. These recursive methods also define iteration spaces, just not *iterative* iteration spaces: each dynamic instance of an invocation of the recursive method is a point in a *recursive* iteration space. In the past five years, there have been some attempts to expand the space of locality-enhancing transformations to work on recursive iteration spaces[1]. For example, Jo and Kulkarni look at transformations on recursive iteration spaces nested within an iterative iteration space, in the context of repeated traversals of trees [20, 21]; the outer loop iterates over points, and the inner recursive traversal of the tree generates a recursive iteration space. Rajbhandari et al. look at *fusing* two (or more) recursive iteration spaces, in an analog of loop fusion [25, 26]; here the separate recursive iteration spaces represent separate traversals of a kd-tree, and the fusion transformation yields a single recursive traversal of the tree.

Up to now, the only locality transformations explored for recursive iteration spaces consider either a recursive space nested within an iterative one, or two or more independent

[1] Note that these efforts do not always conceptualize their work in terms of iteration spaces.

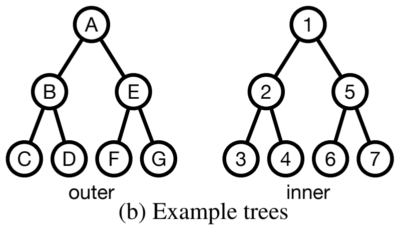
ASPLOS '17, April 08 - 12, 2017, Xi'an, China

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4465-4/17/04…$15.00

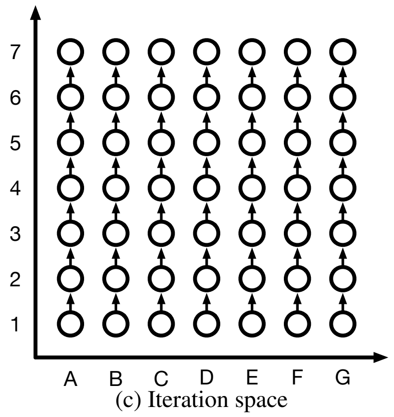
DOI: http://dx.doi.org/10.1145/3037697.3037720

```
1  Node root_o, root_i; //roots of trees

3  recurseOuter(Node o)
4    if (o == null) return;
5    recurseInner(o, root_i);
6    recurseOuter(o.left);
7    recurseOuter(o.right);

9  recurseInner(Node o, Node i)
10   if (i == null) return;
11   join(o.data, i.data);
12   recurseInner(o, i.left);
13   recurseInner(o, i.right);
```

(a) Joining two trees



(b) Example trees



(c) Iteration space

Figure 1: Nested recursion and resulting iteration space

traversals of a recursive space. In this paper, we explore a new class of iteration spaces that target a different class of programs and open new opportunities for transformations: *nested recursive iteration spaces*.

1.1 Nested Recursive Iteration Spaces

Nested recursive iteration spaces arise, unsurprisingly, from nested recursion. Consider the code in Figure 1(a), which performs a join of two trees (imagine, for example, performing a join across two tree-based indices): for each node of the first tree, every node of the second tree is visited to perform the join.

The "work" performed by this nested recursive code is the join operation in Line 11. If the outer tree has n nodes and the inner tree has m nodes, we can see that the join operation is called mn times. If this code is called on the two trees in Figure 1(b), the result is that join will be called 49 times. These 49 dynamic instances of join can be organized in an iteration space in exactly the same manner as iteration spaces that arise from nested loops: each of the two recursive functions represents one dimension of the iteration space, leading to a two-dimensional iteration space. Each point in the iteration space (for brevity, we will call these points "iterations") is labeled by two values: the location in the outer tree the first recursive call (recurseOuter) is at, and the location in the inner tree the second recursive call (recurseInner) is at. Figure 1(c) shows this iteration space. Note that while the labels in this example correspond to specific tree nodes, the recursive calls need not traverse actual trees. In reality, the labels for the iterations correspond to abstract positions in the iteration space.

The arrows in Figure 1(c) represent the order in which the iterations are invoked: for each node of the outer tree (labeled with letters), every node of the inner tree (labeled with numbers) is traversed, with a depth-first, pre-order ordering. In other words, the arrows represent the *schedule* of computation implemented by the original code. We can reason about the locality effects of this particular schedule. There are two important conclusions we can draw. First, there is tremendous reuse in this computation: the computation touches $O(m + n)$ data, but performs $O(mn)$ work. Given this data reuse, there is an opportunity for good locality: if the operations that touch the same portions of the tree are scheduled in close succession, the tree data is likely to remain in cache.

Second, the schedule implemented by the code in Figure 1 is actually quite bad. While the outer tree has good locality (all of the iterations that touch a particular outer tree node are scheduled back to back, so each outer tree node will be brought into cache only once), the inner tree has poor locality. While each inner tree node is touched n times, the accesses occur m iterations apart—the size of the entire inner tree! A simple reuse distance analysis [24] thus indicates that the reuse distance[2] on inner tree nodes is $O(m)$. Hence, if the inner tree gets sufficiently large, *every* access to the inner tree results in a cache miss.

This sort of locality behavior arises in traditional loop-based programs—the join example in Figure 1 is analogous to a vector outer-product in dense linear algebra, and the locality effects on the two vectors in that kernel correspond to the locality effects on the two trees. Loop transformations such as tiling produce a new schedule of computation for vector outer product that has much better locality. But loop tiling is defined for loop-based codes. Is there an analogous schedule that will give good locality for nested, recursive iteration spaces? And can that schedule be efficiently realized in code? This paper answers both of these questions, "yes."

[2] The *reuse distance*, or *stack distance*, of a memory location is the number of unique other locations that are touched between one access to the location and the next access to the same location. Smaller reuse distances mean better locality: roughly, reuse distances smaller than the cache size are likely to be cache hits, and those larger than the cache size are likely to be misses (modulo associativity effects).

1.2 Contributions

This paper makes the following contributions:

1. We develop a simple transformation for nested recursive iteration spaces that is an analog of loop interchange. While this transformation alone does not provide any locality benefits, the conceptual transformation, and the synthesized code that implements the new schedule, form the basis for more complex transformations.
2. We introduce *recursion twisting*, a recursion-friendly analog of loop tiling, and describes how to implement the schedule. Through a reuse distance analysis, we argue that twisted interchange improves locality for nested recursive programs. Interestingly, unlike loop tiling, recursion twisting is parameterless: there is no need to instantiate parameters, such as tile size, based on cache properties. Notably, this means that twisted interchange optimizes for all levels of the memory hierarchy.
3. Unlike loop-based iteration spaces, nested recursive iteration spaces can have highly irregular "loop" bounds, due to irregular truncation conditions in recursion that lead to non-convex bounds. We develop a scheduling strategy to ensure that the bounds of the iteration space are preserved under our transformations, and describe several optimizations to implement these schedules efficiently.
4. We evaluate recursion twisting by examining two simple nested-recursion benchmarks as well as four real-world examples of nested-recursive algorithms: Curtin et al.'s *dual tree* n-body methods [11]. We show, across these six benchmarks, that our transformations can yield between 1.77× and 10.88× speedup (3.94× geomean speedup) over baseline implementations.

1.3 Outline

The remainder of this paper is organized as follows: Section 2 describes the structure of nested recursion that we target, and our foundational scheduling transformation, *recursion interchange*. Section 3 defines recursion twisting. Section 4 explains how our scheduling transformations handle data-dependent recursion truncation. Section 5 describes a compiler transformation that implements recursion twisting. Section 6 evaluates the recursion twisting transformation. Section 7 discusses connections between this work and work on traditional, loop-based iteration spaces. Section 8 describes related work, and Section 9 concludes.

2. Recursion Interchange

This section introduces *recursion interchange*, the foundational transformation that enables the recursion twisting optimization presented in Section 3. First, we explain the template of nested recursion: the control structure of the algorithms that this paper's transformations target.

```
1  TreeNode outer, inner; //binary tree nodes

3  void recurseOuter(TreeNode o, TreeNode i) {
4    if (truncateOuter?(o)) return;

6    recurseInner(TreeNode o, TreeNode i);

8    recurseOuter(o.c1, i);
9    recurseOuter(o.c2, i);
10 }

12 void recurseInner(TreeNode o, TreeNode i) {
13   if (truncateInner1?(i) || truncateInner2?(o, i)) return;

15   work(o, i);

17   recurseInner(o, i.c1);
18   recurseInner(o, i.c2);
19 }
```

Figure 2: General nested recursion template

2.1 Nested recursion template

Figure 2 shows the general template for nested recursion. The template is written in terms of recursion over binary trees, but need not be specific to trees, and need not feature only two recursive calls per recursion.

The recursion template essentially traverses the inner tree for each node of the outer tree (modulo the effects of the truncation methods, which we discuss below). To understand the behavior of the recursion template, it is useful to consider an analogy with a doubly-nested for loop; the two recursive calls are the analog of the two loops. Crucially, almost all of the code in the recursive functions only as the equivalents of loop tests and increments. The code that corresponds to the loop *body* is the call to work in line 15.

The two tree nodes involved in the recursion, o and i, are the equivalent of the loop indices (indeed, this is why the code is not truly specific to trees, as these indices can just be abstract handles, just as loop indices are). The truncation conditions in lines 4 and 13 are the equivalent of the loop termination conditions in a nested for loop: they determine the bounds of the iteration space. Note that the outer recursion is only truncated based on the value of o, while the inner recursion can be bounded based on both o and i—the equivalent of an outer for loop having a termination condition based just on its loop index, while the inner for loop's termination condition can be based on both loops' indices.

Finally, the recursive calls dereference the children of o and i. These are the equivalents of increment operations in for loops. Note that there is no reason there cannot be additional recursive calls in each of the recursions. While here we represent the "increment" operations using field dereferences, these could instead be abstract increment functions that move to the next abstract index, allowing the recursion template to apply to non-tree traversals.

To close the analogy, note that our recursion template is actually a *generalization* of perfectly nested loops: if the template features just one recursive call in each recursion and if the truncation conditions simply test for null, then the recursion template devolves into a doubly-nested loop, with each of the "trees" being linked lists where each node in the two lists represents one value of the corresponding loop index.

Note that truncateInner2? in line 13 allows the inner recursion to be truncated based on both the outer index and the inner index. As we will see in Section 4, this can lead to irregular loop bounds. For the remainder of this section and Section 3, we will assume that truncateInner2? is a no-op (as it is in the tree join example of Figure 1(a)), leading to simple, regular iteration spaces that are easy to reason about.

Assumptions and Terminology The transformations presented in this paper apply whether or not the recursions traverse trees, but our locality analyses presume that the recursions do traverse trees (though the inner and outer recursions may traverse the same tree), as the applications we consider in Section 6 do so. Hence, for simplicity, in the remainder of the paper we will assume that the recursions traverse trees.

For the remainder of the paper, we use the following terminology: the *outer tree* is the tree traversed by the outer recursion of the *original* code (e.g., the tree labeled alphabetically in Figure 1(b)), while the *inner tree* is traversed by the inner recursion of the original code (e.g., the tree labeled numerically in Figure 1(b)). The identity of the two trees is absolute, regardless of the transformation(s) performed on the original code. In contrast, the *outer recursion* is relative to a particular piece of code, and is whichever recursion is the one invoked first, while the *inner recursion* is the recursion invoked from the outer recursion. Hence, in Figure 3, the outer recursion traverses the inner tree, and the inner recursion traverses the outer tree.

2.2 Swapping inner and outer recursions

One of the most basic loop transformations is *loop interchange*, where the inner and outer loop of a doubly-nested loop are swapped. We can imagine an analogous transformation for our recursion template: rather than traversing the entire inner tree for each node of the outer tree, we can traverse the entire *outer* tree for each node of the inner tree. We call this transformation *recursion interchange*.

Recursion interchange is quite straightforward, and the results of applying it to the recursion template are shown in Figure 3 (recall that we are ignoring truncateInner2?). We see that recurseOuterSwapped iterates over the *inner* tree (so the bounds check in line 2 checks the inner tree, and the recursive calls traverse the inner tree) and recurseInnerSwapped iterates over the outer tree. If we were to apply recursion interchange to the iteration space in Figure 1, the resulting iteration is a row-by-row enumeration of the iteration space, instead of column-by-column.

```
1  void recurseOuterSwapped(TreeNode o, TreeNode i) {
2    if (truncateInner1?(i)) return;

4    recurseInnerSwapped(o, i);

6    recurseOuterSwapped(o, i.c1);
7    recurseOuterSwapped(o, i.c2);
8  }

10 void recurseInnerSwapped(TreeNode o, TreeNode i) {
11   if (truncateOuter?(o)) return;

13   work(o, i);

15   recurseInnerSwapped(o.c1, i);
16   recurseInnerSwapped(o.c2, i);
17 }
```

Figure 3: Nested recursion after recursion interchange

Locality effects Recursion interchange does not have substantial effects on locality. Returning to the example from Section 1, where the outer tree has n nodes and the inner tree has m nodes, the original, column-by-column schedule of execution means that the outer tree has $O(1)$ reuse distances, and hence good locality, while the nodes of the inner tree have $O(m)$ reuse distances, and hence poor locality. In the interchanged code, with a row-by-row schedule, the *inner* tree has good locality while the nodes of the *outer* tree have $O(n)$ reuse distances. In practice, for large trees, this means that either version of the code has bad locality.

There is one exception, however: if the trees are sized so that the *outer* tree can fit in cache while the *inner* tree cannot (so $n < m$), then the interchanged code, with its smaller reuse distances, will have good locality while the original code will not. If, instead, it is the inner tree that fits in cache, the original code will feature good locality, while the interchanged code will not. Thus, while recursion interchange alone is typically ineffective, we can exploit this asymmetry—that locality properties are determined by the size of the tree traversed by the inner recursion—to develop a new transformation that *is* effective. We present this transformation next.

3. Recursion Twisting

This section introduces *recursion twisting*, a transformation for nested recursive programs that is the equivalent of a multi-level tiling optimization (indeed, as we note in Section 7, this transformation has some similarities to cache-oblivious algorithms). We present the transformation itself in Section 3.1, analyze the locality implications of performing this transformation on applications that traverse trees in Section 3.2, and discuss its soundness in Section 3.3.

3.1 Transformation

Recall from Section 2.2 that the tree traversed by the outer recursion has good locality, while the tree traversed by the inner recursion has a reuse distance determined by the size of the tree (more accurately, by the size of each traversal). However, if both trees are large, it is immaterial which tree is traversed by the inner recursion, as neither tree fits in cache, and the inner recursion will suffer a large number of cache misses regardless.

However, consider the behavior of the *outer* recursion when it is called on the root of the outer tree in the example from Section 1. In the first invocation of the outer recursion, the entire inner tree is traversed for node A of the outer tree. Then, the outer method is invoked recursively on the left and right children of the root. Each of those invocations can be thought of as a *separate*, *new* nested recursive call, one that starts at node B of the outer tree, and one that starts at node E of the outer tree. Both of these outer recursion calls still traverse the entire inner tree, but the outer (sub)trees that are traversed by the two calls, considered separately, are now smaller. Indeed, after each outer recursive call, the remaining outer tree "rooted" at that outer call gets smaller. Eventually, then, this outer subtree becomes small enough that it can fit in cache.

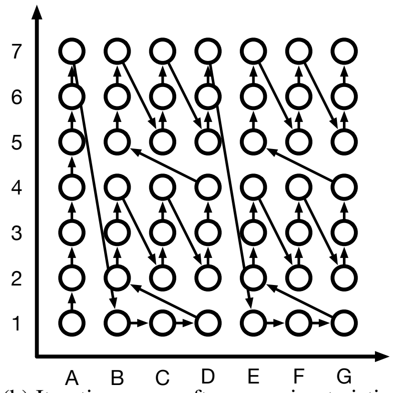
Note that, semantically, recurseOuter (from Figure 2) and recurseOuterSwapped (from Figure 3) are identical: they both perform a "cross product" of two trees: for each node of one tree, visit every node of the other tree. The only difference is which tree is traversed by the outer recursion and which is traversed by the inner recursion. This means that any call to recurseOuter can be replaced with a call to recurseOuterSwapped (and vice versa). Thus, whenever recurseOuter is at a node of the outer tree where its child subtrees are small enough to fit in cache, we can call recurseOuterSwapped on those subtrees, switching the outer tree to the inner recursion, and yielding locality benefits. In other words, even if the outer tree is not small enough to fit in cache initially, after several rounds of recursion, the remainder of the outer tree is small enough that it is worth performing recursion interchange.

So when should this interchange happen? It can be difficult to tell whether a subtree is small enough to fit in cache, so knowing when to switch from traversing the outer tree in the outer recursion to traversing the inner tree in the outer recursion. However, note that it is *always* better to traverse the *smaller* tree in the inner recursion. Moreover, whichever tree is traversed by the outer recursion gets smaller as the recursion progresses, while the inner tree stays the same size. Hence, after continued recursion, the outer tree becomes smaller than the inner tree and it becomes profitable to perform interchange. This process can be repeated, with recurseOuter calling recurseOuterSwapped, and vice versa, when necessary.

Figure 4(a) shows how this schedule can be implemented, with Figure 4(b) showing the resulting execution schedule on our example iteration space. Because the inner and outer trees continually switch places in this schedule, we call this

```
void recurseOuter(TreeNode o, TreeNode i) {
  if (truncateOuter?(o)) return;

  recurseInner(TreeNode o, TreeNode i);

  if (o.c1.size <= i.size)
    recurseOuterSwapped(o.c1, i);
  else
    recurseOuter(o.c1, i);
  if (o.c2.size <= i.size)
    recurseOuterSwapped(o.c2, i);
  else
    recurseOuter(o.c2, i);
}

void recurseOuterSwapped(TreeNode o, TreeNode i) {
  if (truncateInner1?(i)) return;

  recurseInnerSwapped(o, i);

  if (i.c1.size <= o.size)
    recurseOuter(o, i.c1);
  else
    recurseOuterSwapped(o, i.c1);
  if (i.c2.size <= o.size)
    recurseOuter(o, i.c2);
  else
    recurseOuterSwapped(o, i.c2);
}
```

(a) Pseudocode implementing recursion twisting. recurseInner and recurseInnerSwapped remain the same.



(b) Iteration space after recursion twisting

Figure 4: Example of recursion twisting

transformation *recursion twisting*. Each time recurseOuter calls recurseOuterSwap, or vice versa, the schedule "twists."

3.2 Locality analysis

What are the locality implications of recursion twisting? A formal theoretical analysis of the transformation is beyond the scope of this paper, but this section discusses some of the effects of the transformation, using the notions of reuse distance analysis [24]. We analyze a scenario akin to that in our running example: a cross-product of two equally-sized, balanced, binary trees. The general conclusions of this analysis hold for any algorithm where i) the inner and outer recursions traverse trees (that could be the same tree); and ii) the trees are roughly balanced (i.e., the depth of a tree with n nodes is $O(logn)$).

Recursion twisting has good locality Assume that the inner and outer trees have n nodes each, and that work(o, i) accesses exactly node o and node i. In the original nested recursion, the reuse distance of outer tree nodes (disregarding the first access to any node) is $O(1)$ (between successive uses of an outer tree node, only one inner tree node is accessed), while the reuse distance of inner tree nodes is $O(n)$. Hence, the outer tree nodes have good locality, while the inner tree nodes have poor locality. Because each invocation of work accesses an inner tree node and an outer tree node, half the accesses in the program have good locality, and half have bad locality.

In recursion twisting, the root of the outer tree traverses the entire inner tree, and then the two recursions are interchanged. This yields two nested recursions, one called on the inner tree and the left child of the outer tree's root, and the other called on the inner tree and the right child of the outer tree's root. Now, the inner tree has good locality, with reuse distances of $O(1)$, while the outer tree has poor locality—but with reuse distances of $O(n/2)$ (because each outer subtree only has half the nodes). Hence, while half the accesses (on the inner tree) still have good locality, the other half (on the outer tree) has better locality than before. Indeed, after each twist (recursion interchange) in recursion twisting, the tree traversed by the outer recursion will have good locality, and the tree traversed by the inner recursion will have a reuse distance half that of the previous step.

In other words, as the recursion depth gets deeper along both the inner and outer recursion, locality improves rapidly. Because most of the work happens "deep" in the recursion over two trees (for instance, since half of each tree is leaf nodes, fully 1/4th of the work is joining one leaf node with another leaf node), most of the work happens with small reuse distances for *both* trees, and hence the new schedule exhibits good locality.

Examples The effects of recursion twisting can be seen when comparing Figures 1(c) and 4(b). Consider accesses to node 5 of the inner tree, of which there are 7 (one for each node of the outer tree). In the original schedule, the reuse distances for node 5 (counting only tree nodes that are accessed) are, in order of execution, $[\infty, 8, 8, 8, 8, 8, 8]$. In the twisted schedule, the reuse distances are $[\infty, 10, 3, 3, 10, 3, 3]$: half of the accesses have their reuse distances more than cut in half. Note that some of the reuse distances went up, due to the effects of recursion twisting, but the maximum reuse distance is bounded by the size of the two trees and is hence still $O(n)$; moreover, these larger reuse distances merely leave cache misses as cache misses. Most accesses wind up with smaller reuse distances, as "tiles" of execution naturally emerge in the schedule (indeed, $3{\times}3$ tiles are visible in the schedule of Figure 4(b)).

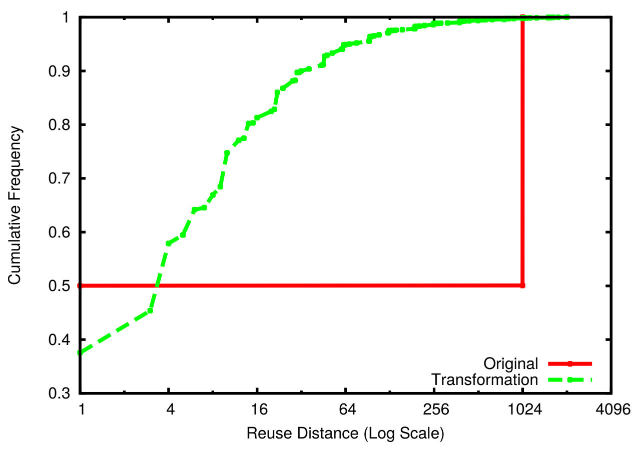


Figure 5: Reuse distances in original vs. transformed code.

To see these effects in a larger setting, Figure 5 shows the results of running a reuse distance simulation on the example from Figure 1(a) with trees of size 1024. The figure shows a CDF plotting the percentage of accesses with reuse distance less than r for all r. We see the hot/cold nature of the original schedule immediately, with 50% of the accesses having a small reuse distance and 50% having a large one. The recursion twisting schedule has an interesting effect: it generally lowers reuse distances, but not uniformly. This is a consequence of the recursive nature of the transformation, with reuse distances halving after each twist. If we were to plot the iteration space of this larger example, we would see a series of *nested* tiles—tiles that are themselves decomposed into tiles—in the schedule, leading to a wider range of reuse distances.

Recursion twisting is parameterless This natural nesting of tiles that emerges from recursion twisting is salutary, as it leads to a *parameterless* transformation. To see why this is beneficial, consider the effect of applying loop tiling to a vector outer product algorithm. As discussed previously, in vector outer product, one vector has good locality, while the second vector has poor locality. If loop tiling with a tile size of b is applied, most elements of the second vector have $O(b)$ reuse distances. If b is smaller than cache, this is excellent—most accesses fit in cache—but if b is larger than cache, then tiling has no effect. Militating against making b arbitrarily small, though, is a second factor: once per tile, the elements of the second vector suffer an $O(n)$ reuse distance, and if there are too many tiles, this results in too many misses. Thus, carefully choosing b is critical to achieving good performance. Indeed, this parameter-driven locality means that not only must the tile size be chosen carefully, but in the presence of multiple levels of cache, multiple levels of tiling must be applied.

In contrast, in recursion twisting, each twist halves the size of the trees involved, so reuse distances naturally, and progressively, shrink, until they become small enough to fit in each level of the cache. Hence, without the aid of any parameters, applying recursion twisting naturally produces good locality for any size cache—and any number of levels of cache. This recursive halving to produce parameter-free, locality-optimized implementations is the same process that many cache-oblivious algorithms leverage [15, 19].

3.3 Soundness

Like with any scheduling transformation, recursion twisting may not always be sound; the transformed code could produce a different result than the original code, as the order in which work methods are called changes. Recursion twisting is sound if, for all pairs of work invocations that have a dependence (they access the same location, and at least one invocation writes to that location), the order they occur in the original code is the same as the order they occur after recursion twisting.

Because recursion twisting changes the schedule of execution in a very complex way, it may seem difficult to determine when the transformation is sound. However, we can establish a fairly simple sufficient condition for soundness:

Claim 1. *Recursion twisting is sound whenever recursion interchange is sound*

Proof sketch A full, formal proof is beyond the scope of this paper. However, intuitively, this claim is true because replacing recurseOuter with recurseOuterSwap (or vice versa) is merely performing recursion interchange on a subset of the iteration space. Furthermore, the entirety of the recurseOuter invocation being replaced with recurseOuterSwap will occur before *any other* iterations are performed. This means that the only iterations that change their order with respect to each other are the ones performed within recurseOuter; everything before still happens before, and everything after still happens after. Thus, if recursion interchange is sound, replacing one instance of recurseOuter with recurseOuterSwap will not change the behavior of the iterations performed by recurseOuter, and cannot change the behavior of any of the other iterations. Hence, a single twist in recursion twisting is sound. A straightforward induction on the number of twists completes the proof.

Implications Note that the general soundness criterion—recursion twisting is sound when recursion interchange is sound—is analogous to the soundness criterion for loop tiling, which is sound whenever loop interchange is sound. While precisely determining when recursion interchange is sound can be difficult (there has been relatively little work in analyzing the structure of dependences for recursive programs [2, 9, 14, 29]), we note that recursion interchange switches a column-by-column execution to a row-by-row execution. This means that while many iterations might change their relative order during recursion twisting, any *intra-traversal* dependences are preserved: any dependences that occur within a single inner recursion in the original code remain within the same column of the iteration space, and hence will be preserved after interchange.

This observation gives rise to a more conservative correctness criterion that is still sufficient to guarantee the soundness of recursion twisting. If the outer recursion is "parallel"—as long as different recurseOuter invocations are independent of each other—the only dependences that can exist are intra-traversal dependences.[3] Hence, if the outer recursion is parallel, recursion interchange is sound, and therefore recursion twisting is sound.

We note that this type of dependence structure is extremely common. Most of the applications we study in Section 6 feature only intra-traversal dependences and parallel outer recursions (the others have no dependences at all). Hence, though this condition is fairly conservative, we find it to be sufficiently precise as a soundness test.

4. Handling Irregular Truncation

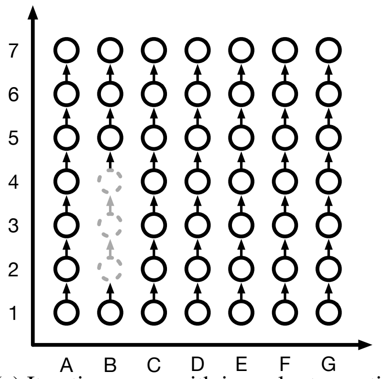
Up until now, we have been considering instances of the nested recursion template in Figure 2 where the truncateInner2? truncation condition is a no-op. This has the effect of ensuring that every instance of the inner recursion traverses the same portion of the tree, regardless of the index of the outer recursion—the only truncation conditions are "local" to the tree being traversed. As a result, the iteration spaces we have considered so far are "rectangular," and, fundamentally, regular (even if they arise from irregular recursion). However, if truncateInner2? is *not* a no-op, then the inner recursion truncation becomes *outer recursion–dependent*, and traversals from different outer recursion indices may look different, leading to irregular iteration spaces and, critically, to correctness challenges for our transformations. This section explains how we deal with these irregular truncation conditions.

4.1 Tracking truncation flags

Figure 6(a) shows an example of an irregular iteration space that might arise from having an outer recursion–dependent truncation. In particular, this is the iteration space that arises from replacing the truncation condition in line 10 of the example in Figure 1(a) with: if $(\mathsf{i} = \mathsf{null} \mid\mid (\mathsf{o.label} = \mathsf{B}\ \&\&\ \mathsf{i.label} = 2))$ return; As we can see, several of the iterations are skipped (and are marked as such with greyed-out circles). Crucially, however, this pattern of skipped iterations is not the same for every outer-recursion index; the iterations are only skipped for index B.

The problem with this truncation is that the truncation *condition* is only checked once, at iteration $(B, 2)$. The other skipped iterations, $(B, 3)$ and $(B, 4)$ are implicitly skipped due to the semantics of recursion. To see why this is a

[3] We leave the development of an analysis to prove the independence of recurseOuter invocations to future work. We note that such an analysis would be similar to Rinard and Diniz's commutativity analysis [27].



(a) Iteration space with irregular truncation

```
1  void recurseOuterSwapped(TreeNode o, TreeNode i) {
2    if (truncateInner1?(i)) return;

4    Set unTrunc = new Set<Node>();
5    recurseInnerSwapped(o, i, unTrunc);

7    recurseOuterSwapped(o, i.c1);
8    recurseOuterSwapped(o, i.c2);
9    foreach (Node n : unTrunc) n.trunc = false;
10 }

12 void recurseInnerSwapped(TreeNode o, TreeNode i, Set<Node>
         unTrunc) {
13   if (truncateOuter?(o)) return;

15   if (truncateInner2?(o, i)){
16     o.trunc = true;
17     unTrunc.add(o);
18   }

20   if (o.trunc != true)
21     work(o, i);

23   recurseInnerSwapped(o.c1, i);
24   recurseInnerSwapped(o.c2, i);
25 }
```

(b) Using truncation flags to implement irregular iteration spaces

Figure 6: Example of irregular iteration space

problem, consider what happens under loop interchange, when the same iteration space is traversed row-by-row, and the outer recursion traverses the inner tree. It is obvious that we can still skip iteration $(B, 2)$ by moving the truncation condition into recursionInnerSwapped and using it to skip the work function. But what truncation condition should be used to skip the other two iterations? Because of the irregular structure of this truncation, we cannot merely skip recursive calls as the original code does. We need to wait until a different outer recursion—visiting a different node of the inner tree—reaches the same point in the outer tree.

Figure 6(b) shows how we can solve this problem. The key is to "remember," when we arrive at iteration $(B, 2)$ that inner tree descendants of 2 need to be skipped whenever the recursion arrives at outer tree node B. We accomplish this by setting a truncation flag in node B (line 16). If an iteration sees the truncation flag set, it skips its work function (line 20)[4]. To ensure that only descendants are skipped, we must also remember to *unset* the truncation flag once all of the children of 2 have finished their traversals. This ensures that when, say, node 5 is traversing the outer tree in the interchanged code, it correctly executes iteration $(B, 5)$. Hence, we maintain a set of outer tree nodes that are truncated by node 2 (line 4), and add any truncated nodes to that set (line 17). After the entire subtree rooted at node 2 is complete, we use that set to reset the truncation flags (line 9).

Crucially, this approach to managing irregular iteration spaces works even in the presence of recursion twisting: when using the "interchanged" order of recursion, this code applies without modification. When using the "regular" order of recursion, we need merely to check whether the truncation flag for the outer node is set, in addition to evaluating truncateInner2? prior to executing recurseInner.

4.2 Optimization: reducing wasted work

Note that the interchanged code cannot use truncateInner2? to cut off any recursion, unlike in the original code, where truncateInner2? truncates the inner recursion. As a result, the interchanged code performs a full cross-product of the two recursive spaces, even if there is data-dependent truncation. This can lead to an explosion in the amount of work performed: dual-tree algorithms (see Section 6) perform a nested recursion of two $O(n)$ size trees, but only have $O(n \log n)$ iterations due to data-dependent truncation. Because the interchanged code cannot cut off any recursion, it is forced to perform $O(n^2)$ iterations.

Recursion twisting ameliorates this work explosion somewhat: whenever the original recursion order is executed—using recurseOuter and recurseInner—data-dependent truncation is able to cut off the inner recursion, saving work. As a result, recursion twisting does not have as much work overhead as recursion interchange.

We can further reduce this overhead through *subtree truncation*: if at any point *every* node in a subtree of the inner tree has the truncation flag set for a particular outer tree node, then the inner tree recursion (performed by recurseOuterSwapped) can be truncated early as well, saving even more work.

When running dual-tree point correlation (see Section 6.1) on a 100,000 point input, the original code performs 1.25 billion iterations. Recursion interchange is forced to perform 5.61 billion iterations, because it cannot truncate any recursions[5]. Recursion interchange, in contrast, performs 1.31 billion iterations, a work overhead of only 4% over the orig-

[4] For clarity, we set the truncation flag in the tree being traversed; if there is no tree being traversed, these truncation flags can be saved in an auxiliary data structure

[5] Note that the interchanged code is still sound: the extra iterations do not perform work, due to the check in line 20 in Figure 6(b).

inal code. Adding subtree truncation leads to 1.27 billion iterations, a work overhead of only 1.8%.

4.3 Optimization: reducing instruction overhead

The truncation code in Figure 6(b) adds *instruction overhead* in addition to work overhead, due to the necessity of unsetting truncation flags (line 9). In addition to requiring a lot of instructions, this loop has cache implications, as outer tree nodes need to be traversed a second time to manipulate the truncation flag.

We can optimize away this instruction overhead when two conditions hold: (i) the original *inner* recursion traverses a tree; (ii) there is only one traversal order for the inner tree, that can be determined *a priori*. The key insight here is that when an outer tree node has its truncation flag set by an inner tree node, the flag is unset when that entire inner subtree is processed. We can number the inner tree nodes according to their traversal order (hence requirement (ii) for the optimization). The truncation flag in each outer tree node is then replaced with a counter, c, with the following semantics: if an inner tree node has number v and $v < c$, then the inner tree node is truncated for that outer tree node.

The counters for the outer tree nodes are initialized as -1 (so there is no truncation). When the truncation flag on an outer tree node would be set by an inner tree node, c is set to the number of the *next* inner tree node after the current inner subtree. Hence, as the inner tree is traversed by recurseOuterSwap, outer tree nodes are naturally "untruncated" as inner subtrees are completed.

5. Prototype Implementation

To help programmers apply recursion twisting, we built a source-to-source transformation tool using Clang's libtooling library.[6] Using annotations, the programmer specifies the two nested recursive functions. The transformation then generates parameterless recursion twisting code, including any necessary truncation code.

Given annotated nested recursive functions, the tool performs a syntactic "sanity check" to make sure that the annotated recursive functions conform to the template shown in Figure 1(a). Next, the tool analyzes the nested recursions to decide whether irregular truncation is performed (in other words, it determines whether any portion of the inner recursion's truncation condition is dependent on the outer recursion). If irregular truncation is needed, it synthesizes twisting-compatible truncation code as described in Section 4.

Our tool is still a prototype, and hence has some limitations. First, our tool does not perform any sort of analysis to prove the soundness of recursion twisting; it relies on the programmer to only annotate nested recursive functions that can be safely transformed. Second, to implement the conditional check that determines when to "twist," our tool assumes that a method can be called to determine the size of the current sub-recursion (i.e., subtree). In the simplest case, this method can simply return the value of a field in the tree data structure being traversed. Finally, while recursion twisting can be applied to recursion with an arbitrary number of recursive calls in each recursive method, the tool currently only works with recursive methods that make two recursive calls.

[6] The tool is available at `https://github.com/kirshanthans/twisted-interchanger`.

6. Experimental Evaluation

We evaluate recursion twisting on two simple benchmarks and four real-world *dual-tree* benchmarks, Curtin et al.'s general approach to n-body algorithms [11]. The basic structure of a tree-based n-body algorithm is that a set of points in space are organized using a spatial tree, such as a kd-tree [4]. This spatial structure is then traversed by each of a set of *query* points to solve the relevant n-body problem, using the tree to prune portions of the search space that will not interact with the query point. In a dual-tree algorithm, the query points are also organized using a spatial tree, which hence becomes the outer tree of a nested recursive algorithm. These algorithms are naturally locality-optimized because the tree organization of the bodies means that similar traversals occur in close succession, reducing reuse distances in the inner tree. We show that recursion twisting can provide substantial performance improvements in scenarios where the baseline dual-tree algorithm has poor locality (i.e., when the inner traversals get large).

6.1 Methodology

Benchmarks We evaluate recursion twisting on six benchmarks:

1. *Tree Join (TJ)* is a cross product of two trees where a pair of nodes contribute to a computation (this benchmark corresponds to Figure 1(a)). We evaluate TJ on trees of size 800 thousand nodes, which the baseline takes 20,189 seconds to complete.
2. *Matrix multiplication (MM)* is a simple computation over two matrices where the outer recursion walks over the rows of the first matrix and the inner recursion walks over the columns of the second. The work(o, i) performed by the nested recursion is a dot product of row o and column i, so the overall computation performs a matrix multiplication. We evaluate MM on matrices of size 40,000 $\times$ 40,000, which the baseline takes 98,232 seconds to complete.
3. *Point correlation (PC)* is a 2-point correlation algorithm that determines how "clustered" a data set is. We evaluate PC on an input of 600 thousand points, which the baseline takes 25,026 seconds to complete.
4. *Nearest neighbor (NN)* find the nearest neighbor of each of a set of query points in a set of data points. We evaluate

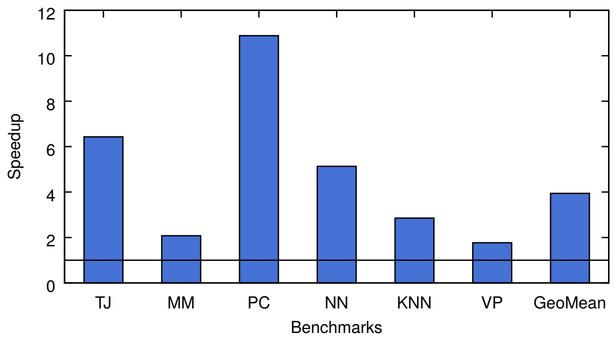
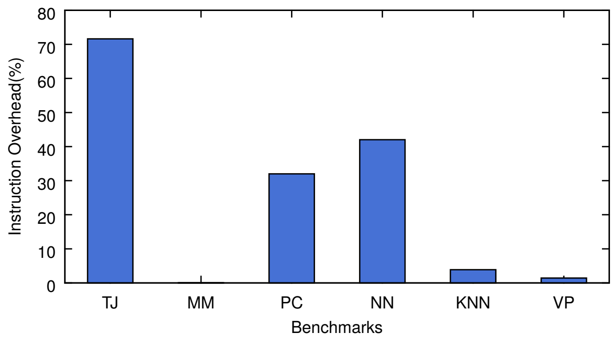
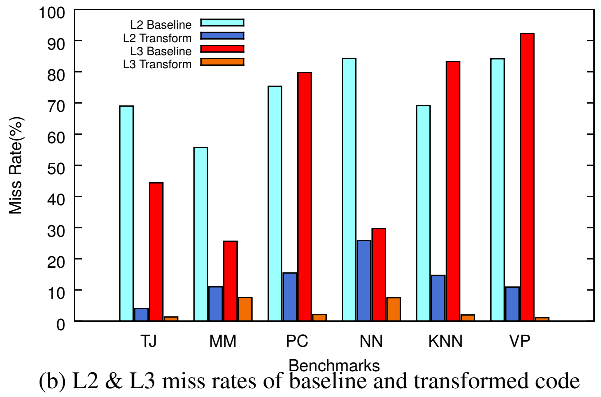


Figure 7: Speedup of recursion twisting



(a) Instruction overhead of transformed code



(b) L2 & L3 miss rates of baseline and transformed code

Figure 8: Performance counter measurements

NN on an input of 1 million points, which the baseline takes 44,868 seconds to complete.

5. *k-Nearest neighbor (KNN)* is like nearest neighbor but finds the k nearest neighbors of each query point. We evaluate KNN ($k = 5$) on an input of 600 thousand points, which the baseline takes 29,758 seconds to complete.

6. *VP-tree (VP)* is a k-nearest neighbor algorithm that uses a vantage point tree instead of a kd-tree. We evaluate VP ($k = 10$) on an input of 400 thousand points, which the baseline takes 122,900 seconds to complete.

TJ and MM have no dependences between iterations, and do not have irregular truncation. PC, NN, KNN and VP—the dual-tree benchmarks—all have dependences carried over the inner recursion (though the outer recursion is still "parallel," satisfying the correctness criterion of Section 3.3), and feature irregular truncation.

Evaluation platform Our evaluation platform is a dual 12-core Intel Xeon with each core running at 2.7 GHz. Each core has 32K of L1 cache, 256K of L2 cache, and each chip shares 20MB of L3 cache. Because we are interested in locality effects, we execute each benchmark in a single thread (note that because of the single-threaded execution, the core has access to the entire last level cache, and hence we require large inputs for the working set to exceed the LLC).

6.2 Performance results

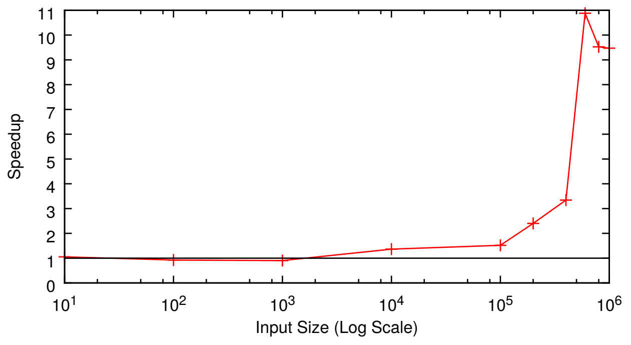
Figure 7 shows the speedup of performing recursion twisting on our baselines. We see that across the board, we get good performance improvements, ranging from 1.77× for VP to 10.88× for PC.

Performance analysis

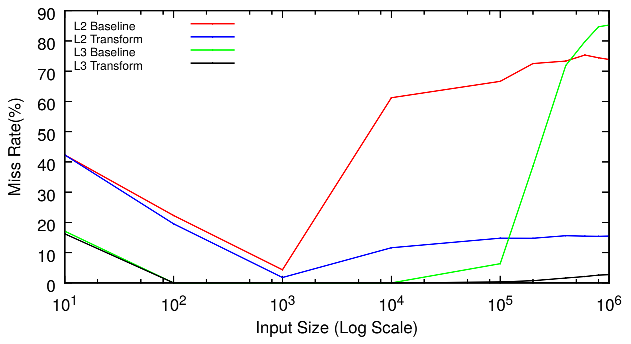
Our performance improvements come from the balance of two factors. First, recursion twisting introduces instruction overhead, due to the overheads of tracking and managing truncation information (see Section 4). Figure 8(a) shows the instruction overhead of recursion twisting, which incurs anywhere from a 1% to a 72% increase in the number of instructions (some of these instruction overheads can be mitigated by introducing cutoffs; see Section 7). Second, recursion twisting yields much better locality. Figure 8(b) shows the L2 and L3 miss rates of the baseline and the transformed code. We can see that miss rates are improved dramatically in both levels of cache, reflecting the fact that recursion twisting targets all levels of cache simultaneously.

Further, we see that for several of our benchmarks, L3 miss rates drop from 80+% to less than 5%. When the input gets large enough, dual-tree algorithms suffer from worst-case locality: since every inner recursion is large enough to trigger capacity misses, *each* inner recursion suffers cache misses on every access, even if there is reuse from one recursion to the next. Recursion twisting, by tiling for all levels of cache, essentially eliminates these capacity misses. We find that the effects on L2 misses are less pronounced (though still evident, as recursion twisting targets all levels of cache). We suspect this is because of other memory accesses that are not targeted by our transformations that hit in the L3 cache despite missing in L2, in both the baseline and our transformed code.

VP also has relatively small performance improvements, but in this case, the baseline for VP has an L3 miss rate of

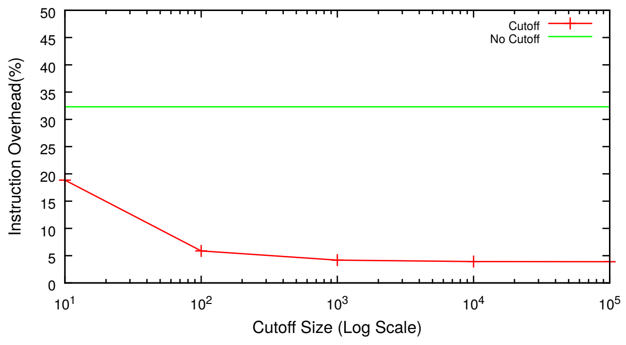


(a) Speedup of recursion twisting

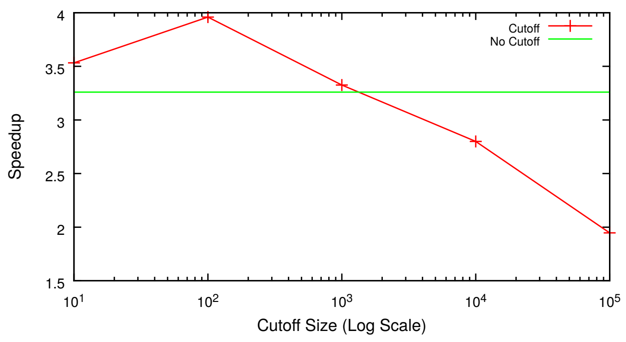


(b) L2 & L3 miss rates of baseline and transformed code

Figure 9: Behavior of PC at different input sizes



(a) Instruction overhead



(b) Speedup

Figure 10: Twisting with cutoff vs. parameterless twisting

92.3%, which recursion twisting is able to reduce to 1.12%! The reason that VP shows relatively little performance improvement despite this dramatic drop in miss rates is that the benchmark is more computationally intensive than, say, PC. The baseline CPI (cycles per instruction) for PC is 6.7—the benchmark is highly memory bound—while the baseline CPI for VP is only 0.93. As a result, while our transformation eliminates many cache misses, there is enough computation to hide much of the effects of those cache misses even in the baseline. We see the opposite effect for TJ: a fairly substantial speedup despite the high instruction overhead and the lower baseline L3 miss rate. Here, since TJ has low computational intensity, almost all of the time is spent fetching tree data, exactly the memory accesses our transformations target.

Effects of input scale

Finally, we note that the gains of recursion twisting increase with input size, though they eventually level off. Figure 9(a) shows the speedup of PC for a range of input sizes (note the log-scale x-axis), and Figure 9(b) shows L2 and L3 miss rates. For small inputs, there is virtually no speedup, or even a slowdown. Recall that the reuse distance of the baseline algorithms is determined by the sizes of the inner recursions. For small inputs those recursions fully fit in cache (even modulo the difference from one traversal to the next), and hence the baseline has good locality (indeed, the L3 cache miss rate for PC is effectively 0 until an input of 100K). Hence, performance is dominated by the instruction overhead of recursion twisting. (Note, by the way, that at the very smallest input sizes, miss rates are *higher* than at somewhat larger sizes; this is because at extremely small sizes, the effects of compulsory misses to bring the trees into cache are more noticeable.)

As the input size grows, PC begins to suffer L3 cache misses, and its speedup commensurately increases. Eventually, the inner recursions get so large that the caches are saturated, and the L3 miss rate levels off (at about 80%): the vast majority of accesses to L3 cache are misses. At this point, recursion twisting is able to eliminate virtually all misses that are targeted by the transformation, yielding an L3 miss rate of about 3%. Because there is no more opportunity to eliminate misses, the speedup also levels off.

7. Discussion and Future Work

7.1 The price of parameterlessness

As the previous section shows, recursion twisting can yield large performance improvements when the initial application has poor locality (i.e., whenever the inner recursion dimension is large, leading to high reuse distances in the baseline). However, if the baseline has good locality already, recursion twisting cannot improve locality further, so instead only adds overhead. Even in situations where there is a locality improvement to be had, the cost of so much recursion still introduces instruction overhead relative to the baseline (especially due to tracking truncation effects, as discussed in Section 4). Both of these overheads are, essentially, the price of being parameterless. While avoiding any cache-specific transformation parameters is good for targeting all levels of the memory hierarchy, it means that the overhead of twisting continues to be paid even after there is no longer any locality benefit to gain. If, instead, there were *cutoff* parameters that would switch back to the standard recursive schedule, sans twisting, whenever the inner tree traversals fit in cache, we could eliminate most of the instruction overhead without paying any locality penalties[7]. The challenge, of course, is determining what this cutoff parameter should be: cut off too early and the inner traversals will not fit in cache, precluding any locality benefit; cut off too late and much of the benefit of providing a cut-off parameter is lost. The problem of determining an appropriate cutoff is complicated by any irregularity in the nested recursion, as such irregularity makes predicting traversal sizes, and hence the optimal cutoff, more difficult.

To investigate the impact of cutting off recursion twisting, we modified the recursion twisting implementation of PC to include a cutoff parameter: the twisting code will only switch from the original recursion order to the interchanged order if the inner tree size is greater than the cutoff parameter. Figure 10(a) shows the effect that implementing cutoff has on instruction overhead, for different cutoff values, as well as the instruction overhead of the parameterless recursion twisting implementation for comparison. As expected, implementing cutoff reduces instruction overhead, as twisting does not occur for small subtrees. Note, too, that as expected, instruction overhead is higher for smaller cutoff parameters. Nevertheless, for all the values we studied, adding cutoff results in lower instruction overhead than the baseline parameterless version.

Figure 10(b) shows how this reduction in instruction overhead interacts with locality effects to affect speedup. (Note that we use a smaller input for PC than in the experiments of Section 6, so the speedup of the baseline parameterless version is lower.) If the cutoff value is too large, we get less locality improvement so, despite the lower instruction overhead, speedup is worse than the parameterless version. Smaller cutoff values can produce better speedup than the baseline, but note that the smallest cutoff value does not yield the best speedup—the higher instruction overhead reduces performance. Overall we see that setting a cutoff parameter correctly *can* produce higher speedup (as expected), but the parameterless version is not too far off from the best cutoff version. Investigating how to set the cutoff parameter correctly in recursion twisting is an interesting avenue of future work.

7.2 Cache obliviousness, loop nests, and multi-level loop tiling

There is an intriguing connection between recursion twisting and cache-oblivious algorithms. As we noted above, both exploit divide-and-conquer strategies to keep shrinking the working set of the algorithm until it fits in cache. By continuing to divide the working set, the resulting algorithm provides multiple nested working sets, such that each cache is exploited (asymptotically) optimally [15, 19]. While we have not proven a similar property for recursion twisting, the structure of the schedule that twisting generates seems quite similar, and pursuing this line is an interesting avenue of future work.

One hint of the close connection between recursion twisting and cache-obliviousness is to consider the way in which languages like Cilk handle for loops [10]: rather than being executed iteratively, the loops are translated into a divide-and-conquer recursion (often with a granularity cutoff). We can take a doubly-nested loop program—say matrix-vector multiplication—and translate both loops into this divide-and-conquer form. Applying recursion twisting to resulting nested recursion automatically yields something similar to the cache-oblivious implementation!

Indeed, recursion twisting may provide a simple way to automatically apply the benefits of multi-level loop tiling to nested for loops, with (i) a much simpler code structure than a deep tiled loop nest; and (ii) a schedule that fits all levels of cache without knowing the number and sizes of caches.[8] Another useful direction of future work is to generalize recursion twisting to more than two levels of recursion, to allow it to handle algorithms like matrix-matrix multiplication.

7.3 Parallelism

We have not evaluated parallel implementations of any of our benchmarks. However, adding parallelism to nested recursion is completely straightforward. Recall from Section 3.3 that a sufficient condition for the soundness of recursion twisting is if each outer recursive step is independent of the rest. This independence means that the outer recursions can be executed in a task-parallel manner, using annotations such as Cilk's `spawn` [16], with each outer recursive call executed independently of the others. At any point in the process, recursion twisting can be applied to a `spawn`ed task to improve its locality. Note, however, that once recursion

[7] We note that cache-oblivious algorithms face a similar challenge: the divide-and-conquer recursive structure of these algorithms yield the cache-oblivious property, but add substantial overhead. Adding cutoff parameters, and hence cache-awareness, is a standard trick to improving performance [31]

[8] Indeed, with this extension, recursion twisting can be considered a generalization of Yi et al.'s approach to generating divide-and-conquer schedules for affine loop nests [30]; see Section 8 for more discussion of this connection.

twisting is applied, it is no longer sound to treat outer recursions as independent of one another—the recursive calls can no longer safely be spawned—so twisting should only be applied once enough parallelism has been generated.

8. Related Work

As discussed in the introduction, most work in the space of locality transformations focuses on *iterative* iteration spaces—spaces that arise from for loops with well-defined bounds [1]. Both the unimodular framework [3] and the polyhedral framework [5, 12, 13] target such for loops, enabling the use of transformations such as loop interchange, loop skewing, loop fusion, and loop tiling. While these types of transformations are widespread, they do not handle recursive programs.

In recent years, there has been more work targeting locality transformations for recursive programs, including dynamic pointer alignment [32], point blocking [20], traversal splicing [21]. These transformations are, effectively, variants of loop tiling in various ways for recursion that is nested within a normal, iterative iteration space (specifically, for loops). Thus, while these transformations handle *single* recursive iteration spaces, they do not handle nested recursive iteration spaces. Moreover, the transformations are *not* parameterless: they require setting parameters based on cache size to ensure that the locality benefits are obtained.

More recently, Rajbhandari et al. have proposed a transformation that combines kd-tree traversals into a single traversal of the tree structure [25, 26], which has locality benefits somewhat similar to point blocking [20]. In some sense, this transformation targets multiple recursive iteration spaces—the multiple kd-tree traversals—but there are some key differences between its approach and recursion twisting: (i) the recursions are sequential, not nested; (ii) the recursions have to be "matched"—they traverse the same tree; (iii) the transformation targets a statically-bounded number of such recursions, rather than the unbounded number of recursions that can arise from nested recursion (whether nested within an iterative or recursive iteration space).

Complementary to our approach, several prior works have looked at *layout transformations* for tree data structures [6–8, 17, 28]. These techniques focus on changing how trees are organized in memory, ensuring that, for example, nodes from the same subtree are allocated to the same cache line. As a result, traversals of a tree are more likely to enjoy *spatial* locality. We expect these types of transformations to be complementary to recursion twisting, which instead focuses on *temporal* locality.

One possible approach to handling nested recursive programs is to use program transformations to *eliminate* the recursion, yielding new programs that operate over loops, instead [18, 22]. While this approach has been used to perform some performance optimizations [23], to our knowledge, there has been no efforts to use recursion elimination to enable loop transformations.

Yi et al. propose a technique that takes programs featuring nested loops and transforms them into programs that instead use (non-nested) recursion [30]. This new recursive structure schedules the original (affine) iteration space in a divide-and-conquer manner to enhance locality. Interestingly, the iteration spaces and schedules that arise from their transformations bear a strong resemblance to the types of schedules that arise from recursion twisting. There are two key differences between our techniques, both arising from the fact that we *start* with recursive codes, rather than iterative ones. First, the recursive code Yi et al. generate pushes all the work into the "leaves" of the recursion—the recursive "spine" of the code is simply used to schedule the underlying affine iteration space. In contrast, the kinds of programs that arise from nested recursion can perform work during the recursive steps as well, resulting in more complicated schedules. Second, the iteration spaces we consider in recursion twisting can be more irregular than affine iteration spaces due to irregular truncation, necessitating a more complex transformation (see Section 4).

9. Conclusions

Most work on locality enhancing scheduling transformations has focused on programs that feature regular for loops. There has been relatively little work on transforming programs that involve recursion, and what work there has been has focused on either non-nested recursion or recursive traversals nested in for loops. We developed a new transformation called recursion twisting that targets recursions nested within other recursions. This pattern arises in algorithms such as dual-tree n-body codes and in recursive decompositions of nested loops. Recursion twisting is a parameter-free locality transformation that simultaneously enhances locality for all levels of cache, and showed that it can provide substantial performance benefits. This work represents the first generalization of transformations for nested, iterative iteration spaces to nested, *recursive* iteration spaces, and is a necessary step towards producing a new, general transformation infrastructure that can simultaneously handle iterative and recursive control structures.

Acknowledgments

The authors would like to thank the anonymous referees for their helpful comments and suggestions. This work was supported in part by the following grants: NSF CCF-1150013, NSF CCF-1439126, and DOE DE-SC0010295.

References

[1] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[2] Pierre Amiranoff, Albert Cohen, and Paul Feautrier. Beyond Iteration Vectors: Instancewise Relational Abstract Domains. In *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, pages 161–180, 2006.

[3] Utpal Banerjee. Unimodular Transformations of Double Loops. In *Languages and Compilers for Parallel Computing*, 1991.

[4] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, September 1975.

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[6] Trishul M. Chilimbi, Bob Davidson, and James 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.

[7] Trishul M. Chilimbi, Mark D. Hill, and James 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.

[8] Trishul M. Chilimbi and James 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.

[9] Albert Cohen and Jean-François Collard. Instance-Wise Reaching Definition Analysis for Recursive Programs Using Context-Free Transductions. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 332–, Washington, DC, USA, 1998. IEEE Computer Society.

[10] Intel Corp. Intel Cilk Plus Language Extension Specification, 2011.

[11] Ryan R. Curtin, William B. March, Parikshit Ram, David V. Anderson, Alexander G. Gray, and Charles L. Isbell. Tree-Independent Dual-Tree Algorithms. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 1435–1443, 2013.

[12] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem. Part I. One-dimensional Time. *International Journal of Parallel Programming*, 21(5):313–347, 1992.

[13] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[14] Paul Feautrier. A Parallelization Framework for Recursive Tree Programs. In *Euro-Par '98 Parallel Processing, 4th International Euro-Par Conference, Southampton, UK, September 1-4, 1998, Proceedings*, pages 470–479, 1998.

[15] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-Oblivious Algorithms. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 285–298, 1999.

[16] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[17] Hwansoo Han and Chau-Wen Tseng. Exploiting Locality for Irregular Scientific Codes. *IEEE Trans. Parallel Distrib. Syst.*, 17(7):606–618, 2006.

[18] Stefaan Himpe, Francky Catthoor, and Geert Deconinck. Control Flow Analysis for Recursion Removal. In *Software and Compilers for Embedded Systems, 7th International Workshop, SCOPES 2003, Vienna, Austria, September 24-26, 2003, Proceedings*, pages 101–116, 2003.

[19] Hong Jia-Wei and H. T. Kung. I/O Complexity: The Red-blue Pebble Game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, pages 326–333, New York, NY, USA, 1981. ACM.

[20] Youngjoon Jo and Milind Kulkarni. Enhancing Locality for Recursive Traversals of Recursive Structures. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 463–482, New York, NY, USA, 2011. ACM.

[21] Youngjoon Jo and Milind Kulkarni. Automatically Enhancing Locality for Tree Traversals with Traversal Splicing. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 355–374, New York, NY, USA, 2012. ACM.

[22] Yanhong A. Liu and Scott D. Stoller. From Recursion to Iteration: What Are the Optimizations? In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '00, pages 73–82, New York, NY, USA, 1999. ACM.

[23] Yanhong A. Liu and Scott D. Stoller. Dynamic Programming via Static Incrementalization. *Higher-Order and Symbolic Computation*, 16(1-2):37–62, 2003.

[24] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[25] Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noel Pouchet, Fabrice Rastello, Robert J. Harrison, and P. Sadayappan. A Domain-specific Compiler for a Parallel Multiresolution Adaptive Numerical Simulation Environment. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 40:1–40:12, Piscataway, NJ, USA, 2016. IEEE Press.

[26] Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, Robert J. Harrison, and P. Sadayappan. On Fusing Recursive Traversals of K-d Trees. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 152–162, New York, NY, USA, 2016. ACM.

[27] Martin C. Rinard and Pedro C. Diniz. Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers. *ACM Trans. Program. Lang. Syst.*, 19(6):942–991, November 1997.

[28] 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.

[29] Yusheng Weijiang, Shruthi Balakrishna, Jianqiao Liu, and Milind Kulkarni. Tree Dependence Analysis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 314–325, New York, NY, USA, 2015. ACM.

[30] Qing Yi, Vikram Adve, and Ken Kennedy. Transforming Loops to Recursion for Multi-level Memory Hierarchies. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 169–181, New York, NY, USA, 2000. ACM.

[31] Kamen Yotov, Tom Roeder, Keshav Pingali, John Gunnels, and Fred Gustavson. An Experimental Comparison of Cache-oblivious and Cache-conscious Programs. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 93–104, New York, NY, USA, 2007. ACM.

[32] Xingbin Zhang and Andrew A. Chien. Dynamic Pointer Alignment: Tiling and Communication Optimizations for Parallel Pointer-based Computations. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '97, pages 37–47, New York, NY, USA, 1997. ACM.