Tree Dependence Analysis

Yusheng Weijiang Shruthi Balakrishna Jianqiao Liu Milind Kulkarni

School of Electrical and Computer Engineering
Purdue University, USA
{yweijian, balakrs, liu1274, milind}@purdue.edu

Abstract

We develop a new framework for analyzing recursive methods that perform traversals over trees, called *tree dependence analysis*. This analysis translates dependence analysis techniques for regular programs to the irregular space, identifying the structure of dependences within a recursive method that traverses trees. We develop a dependence test that exploits the dependence structure of such programs, and can prove that several locality- and parallelism-enhancing transformations are legal. In addition, we extend our analysis with a novel path-dependent, conditional analysis to refine the dependence test and prove the legality of transformations for a wider range of algorithms. We then use these analyses to show that several common algorithms that manipulate trees recursively are amenable to several locality- and parallelism-enhancing transformations. This work shows that classical dependence analysis techniques, which have largely been confined to nested loops over array data structures, can be extended and translated to work for complex, recursive programs that operate over pointer-based data structures.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

Keywords dependence analysis, irregular algorithms, loop transformations

1. Introduction

Many *dependence analysis* techniques have been developed to determine when applying loop transformations—such as loop interchange, fusion and tiling [2]—to *regular* programs—array programs with affine loop bounds and index expressions—is legal [1, 4, 5, 10, 17, 20, 31, 32]. While there have been many attempts to extend these transformations to handle more sophisticated programs, including those that have non-affine loop bounds and index expressions [21, 28, 29], these tools have largely been confined to array programs using nested loops.

In recent work, Jo and Kulkarni developed an optimization called *point blocking* that performs loop tiling–like transformations not on nested loops, but instead on *repeated recursive traversals* of pointer-based tree structures [15]. Point blocking works by grouping together multiple traversals of a tree into a block and performing a single traversal of the tree. At each node of the tree, all traversals that must perform computation at that tree node do their work before the block moves on to the next node of the tree. In essence, the computations performed by multiple traversals are reordered to promote locality in the tree.

Unfortunately, while this transformation resembles loop tiling (see Section 2.2), existing dependence analyses cannot be applied, as point blocking targets pointer-based, recursive programs. Instead, Jo and Kulkarni establish the legality of their transformations through a simple, sufficient condition: their transformations can be applied when the traversals over the tree structure are independent of each other.

However, this sufficient condition misses many optimization opportunities. Consider inserting a set of points into a binary search tree, as shown in Figure 1(a). Point blocking can be correctly applied to the code, as shown in Figure 1(b), even though there is clearly a dependence from one traversal to the next, as each insertion changes the tree. The reason for this is that if multiple points in a block travel down the same path of the tree, and the first point in the block inserts a node into the tree, subsequent points in the block see the *new* node that was inserted, as they would have in the original code. The dependence is preserved! *This pattern of behavior is quite common*, arising in, for example, top-down tree building algorithms for building kd-trees and Barnes-Hut octrees. Handling such cases requires a more sophisticated notion of what kinds of dependences preclude point blocking.

```
tree = /* bst */
vals = /* ints > 0 */
foreach (i in vals)
  recurse(tree, i)

recurse(n, i)
  if (n.val = -1)
    n.val = i; return;
  if (n.val < i)
    if (n.l = null)
      n.l = new node;
      n.l.val = -1;
    recurse(n.l, i)
  else
    if (n.r = null)
      n.r = new node;
      n.r.val = -1;
    recurse(n.r, i)
```

(a) BST insertion code

```
// i[], l[], r[] = blocks
foreach (i[] in vals)
  recurse(tree, i[])

recurse(n, i[])
  if i.size = 0
    return;
  foreach (j in i)
    if (n.val = -1)
      n.val = j; continue;
    if (n.val < j)
      if (n.l = null)
        n.l = new node;
        n.l.val = -1;
      l[].append(j);
    else
      if (n.r = null)
        n.r = new node;
        n.r.val = -1;
      r[].append(j);

  recurse(n.l, l[]);
  recurse(n.r, r[]);
```

(b) Blocked BST code

Figure 1: BST insertion, unblocked and blocked.

PLDI'15, June 13–17, 2015, Portland, OR, USA
ACM. 978-1-4503-3468-6/15/06
http://dx.doi.org/10.1145/2737924.2737972

Contributions In this paper, we present a *tree dependence analysis*, which provides a more sophisticated picture of the dependences in a tree-traversal program. Analogous to array dependence analyses, which allow complex loop transformations to be performed even if there are loop-carried dependences, our tree dependence analysis provides enough information to allow restructuring transformations like point blocking to be performed even in the presence of dependences between traversals. The contributions we make are:

- A novel dependence test that can prove the legality of point blocking even in the face of complex dependences (Section 3), and a proof of the soundness of point blocking under this test.
- An analysis that applies our dependence test to tree-traversal programs (Section 5). While shape analyses can often determine whether there are dependences between accesses to recursive data structures [18], our analysis reveals the *structure* of these dependences with respect to the recursive control flow of the program.
- A refinement of our dependence analysis that uses *path conditions* to prove that certain dependences that appear to exist can never arise during an execution (Section 6).
- An experimental evaluation that shows our analysis enables significant performance improvements from three transformations: point blocking, *traversal splicing* [16], and a transformation that automatically derives parallel tree construction implementations from their sequential specification.

This paper presents, to our knowledge, the first attempt to lift the kinds of sophisticated dependence analysis techniques developed for programs that loop over arrays to more complex programs that manipulate pointer-based data structures, enabling a host of locality- and parallelism-enhancing transformations to be applied to recursive tree programs.

2. Background and Motivation

This section discusses the theory of loop transformations for array programs—specifically, interchange, which enables tiling—and then summarizes recent work by Jo and Kulkarni that develops analogous tiling transformations for trees.

2.1 Loop Transformations for Array Programs

Perhaps the most popular locality-enhancing transformation for loops over arrays is *loop tiling*, which transforms a double-nested loop into a triple- (or quadruple-) nested loop [17] , as in the following abstract example:

```
for (i := 0; i < N; i ++)
 for (j := 0; j < N; j ++)
   A[f1(i)][f2(j)] = ...;   ... = A[g1(i)][g2(j)]
```

Becomes:

```
for (ii := 0; ii < N; ii  += B)
 for (j := 0; j < N; j ++)
   for (i := ii; i < ii + B; i ++)
     A[f1(i)][f2(j)] = ...;   ... = A[g1(i)][g2(j)]
```

The legality of tiling boils down to whether *loop interchange* is legal [32]; if the inner and outer loop of the above example can be swapped, then loop tiling is legal.

Determining whether loop interchange is legal requires understanding how interchange affects the behavior of the loop. Conceptually, loop interchange is a *rescheduling* of the loop iterations. The original loop consists of an *iteration space*—dynamic instances of the loop body, each with a different value of i and j—that is totally ordered: $(i_1, j_1) \prec (i_2, j_2) \Leftrightarrow (i_1 < i_2) \vee ((i_1 = i_2) \wedge (j_1 < j_2))$. Loop interchange moves the j loop to the outside, producing a different total ordering of the same iteration space: $(i_1, j_1) \prec (i_2, j_2) \Leftrightarrow (j_1 < j_2) \vee ((j_1 = j_2) \wedge (i_1 < i_2))$.

When is this rescheduling legal? Answering this question requires understanding the dependence structure of the loop [1]. If, in the original schedule, one iteration of the loop, (i_1, j_1), writes to a location that a later iteration, (i_2, j_2) reads from, we must ensure that the new schedule does not exchange the order of these two iterations, which would result in the second iteration reading the wrong value. The following *dependence test* captures the conditions under which loop interchange is legal.[1]

$$\begin{aligned} \nexists\, i_1, i_2, j_1, j_2 \,.\, f_1(i_1) = g_1(i_2) \wedge f_2(j_1) = g_2(j_2) \wedge \\ (i_1 < i_2 \wedge j_1 > j_2) \end{aligned} \tag{1}$$

The first line of the test captures whether a pair of iterations access the same location, while the second line of the test captures whether those iterations will execute in a different order after interchange.

Sophisticated dependence analyses such as the Omega test [20] and compilers such as PLuTo [5] use integer linear programming–based techniques to prove that interchange is legal. These analyses rely on the fact that in most array programs, the indexing expressions f_1, f_2, g_1, and g_2 are affine, and hence amenable to ILP. As a result, a long standing open problem has been whether similar tiling techniques exist for non-affine, non-loop-based programs, and how to prove their legality.

2.2 Loop Transformations for Trees

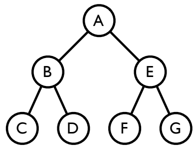
In recent work, Jo and Kulkarni developed a locality-enhancing transformation called *point blocking* for programs that repeatedly traverse tree data structures [15]. Figure 2(b) shows abstracted pseudocode capturing the general structure of these algorithms: each of a series of points (a structures capturing a single traversal's data) recursively traverses a tree. As each point accesses the same tree, there is data reuse in the algorithm, and an opportunity to exploit locality if multiple points' operations on the same data can be brought closer together.

The key insight behind point blocking is that the tree-traversal algorithm can be abstracted as a loop nest, with the loop over the points as the outer loop and the recursive traversal as the inner "loop." Each "iteration" in this abstraction consists of the recursive method body being executed by a particular point at a particular node of the tree; the recursion and pointer-chasing merely serve to determine the order in which the nodes are visited.

Figure 2(c) shows an example iteration space and total order for a series of recursive traversals of the tree shown in Figure 2(a). The x-axis represents the points that traverse the tree, while the y-axis represents the nodes visited by the point. Note that some of the iterations are greyed out, and the traversal skips past them. A traversal may not visit the entire tree—it may be truncated and skip visiting a subtree.

Given this iteration space abstraction, Jo and Kulkarni describe a "loop interchange" transformation, with the total order shown in Figure 2(e). This has an analogous reordering effect as loop interchange in the regular iteration spaces produced by array programs; in the interchanged code, every point visits a particular node in the tree before moving on to the next node in the tree. Point blocking is a combination of strip mining the point loop (breaking the point loop into a series of smaller loops that operate over subsets of points) and then interchanging the inner point loop with the traversal loop. This is a direct analog of strip mining + interchange, a common technique for tiling array programs [32].

[1] In a full dependence test, there are additional constraints to ensure that both iterations fall within the bounds of the loop nest; we ignore these constraints for simplicity.

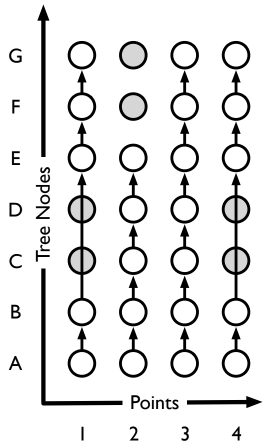


(a) Example tree for traversal

```
tree = /* tree root */
points = /* points */
foreach (p in points)
  recurse(tree, p)

recurse(n, p)
  if truncate?(n, p)
    return;
  if isleaf?(n)
    return;
  /* do work */
  recurse(n.left, p)
  recurse(n.right, p)
```

(b) Pseudocode for traversal



(c) Iteration space before interchange/point blocking

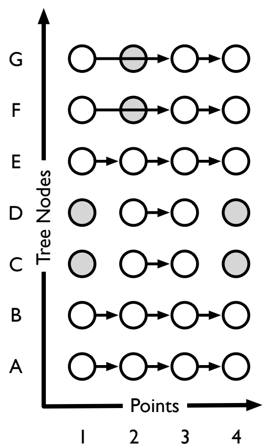
```
tree = /* tree root */
points = /* points */

//bp is block of points
foreach (bp in points)
  recurse(tree, bp)

recurse(n, bp)
  if isempty?(bp)
    return;
  foreach (p in bp)
    if truncate?(n, p)
      continue;
    if isleaf?(n)
      continue;
    /* do work */
    //add to next block
    nb.add(p)

  recurse(n.left, nb)
  recurse(n.right, nb)
```

(d) Blocked traversal



(e) Iteration space after interchange/-point blocking

Figure 2: Point blocking

Figure 2(d) shows the transformed code. Instead of the recursive method operating on a single point, it operates on *blocks* of points. After each point in the block interacts with a particular node, the points that want to continue traversal are added to a "next" block, which continues down the tree. If a block is empty, that means no points want to visit a particular node (or subtree), so the traversal is truncated.

"Multi callset" traversals In the examples of Figure 2, there is a single linearization of the nodes of the tree, and each point's traversal is some subsequence of that linearization. Hence, when points are placed into a block, the order that the block traverses the tree is the same as the traversal orders of any of the individual points. Jo and Kulkarni call these "single callset" traversals [16]. However, some algorithms, such as nearest neighbor, have point-dependent traversal orders: different points traverse the tree in different orders; these are known as "multi callset" traversals. In this paper, we only concern ourselves with single call set traversal algorithms, as they are the only ones that admit a sophisticated dependence test. Multi callset algorithms can still be analyzed using a test for independence.

3. Point Blocking Legality

This section lays out a dependence test for point blocking. For brevity, we use "iteration" to refer to the operation(s) performed by a single point at a single tree node.

3.1 A Conservative Approach

Jo and Kulkarni noted that despite the rescheduling imposed by point blocking, each point still traversed the tree in the same order as before [15]. Hence, any dependences carried over the "traversal loop" but not over the "point loop" would be preserved. Thus, they applied point blocking whenever the enclosing point loop was parallelizable, ensuring that any dependences were only carried across the traversal loop. This criterion is too conservative. Not all point loop–carried dependences are violated by point blocking, as in the BST-insertion example from Figure 1.

3.2 A Dependence Test for Point Blocking

To develop a more accurate dependence test for tree codes, we consider the two clauses of the dependence test for array programs in Equation 1. The first clause picks out the existence of iterations that have a dependence. If only that clause were in the dependence test, then any loop-carried dependence would preclude loop interchange. It is the second clause of the test (on the second line) that provides the precision: a loop carried dependence is only a problem if the second iteration (*i.e.*, the (i_2, j_2) iteration) encounters the dependence earlier in the j loop than the first iteration.

The iteration space diagrams of Figures 2(c) and 2(e) give us some insight into what an analogous dependence test for point blocking might look like. Each "iteration" in a traversal code is identified by a point/node pair: (p, n). Suppose there is a dependence between the traversal executed by point p_1 and a later point p_2: p_1 accesses a location in the tree when it is visiting node n_1, and p_2 accesses the same location in the tree when it is visiting node n_2, with at least one of the accesses being a write. This dependence is preserved by point blocking if n_2 is the same as n_1 (both points are at the same node when the dependence occurs) or n_2 is later in the traversal order than n_1.

To formalize this dependence test, let us label each statement that reads or writes a location in the recursive method body as $s_1, s_2, \ldots$. Because the particular location read or written by a statement depends on where in the tree the recursive method is, we specify the location being accessed by statement i during iteration (p, n) as $s_i(p, n)$.

Making a recursive call requires accessing the arguments to the recursive call. Because point blocking defers making recursive calls until after all points in the block execute the rest of the method body, it makes sense to treat the read(s) performed as part of the method invocation as part of the *next* iteration performed by the point. This is easily handled by assuming there are dummy statements at the beginning of the method body that read the arguments to the method.

Two dynamic statements, $s_i(p_i, n_i)$ and $s_j(p_j, n_j)$ *interfere* (written $s_i(p_i, n_i) \bowtie s_j(p_j, n_j)$) when they access the same location and one of the statements is a write. Note that just because a statement exists in a recursive method body does not mean that every point will execute that statement at every node of its traversal. We thus define an execution-based interference operator, $\bowtie_e$, which adds the condition that statement s_i executes when point p_i is visiting node n_i.

We can now define a dependence test under which point blocking is legal; note the similarity to Equation 1:

$$\begin{aligned} \nexists\, p_i, p_j, n_i, n_j, s_i, s_j \,.\, & s_i(p_i, n_i) \bowtie_e s_j(p_j, n_j) \wedge \\ & (p_i \prec p_j \wedge n_i \succ n_j) \end{aligned} \tag{2}$$

Theorem 1. *If Equation 2 is satisfied for a recursive traversal program, then applying point blocking to the program will not break any dependences.*

Proof. We proceed by contrapositive: we assume that applying point blocking to the program breaks dependences, and show that therefore the dependence test must be violated.

For a dependence to be broken, one must exist in the first place. Hence, let (p_i, n_i) and (p_j, n_j) be the two dependent iterations, with $(p_i, n_i) \prec (p_j, n_j)$. We thus have $s_i(p_i, n_i) \bowtie_e s_j(p_j, n_j)$. In the original program, a point's traversal is completed before moving on to the next point. Hence, $p_i \prec p_j$. Note that if, after applying point blocking, p_i and p_j are placed in different blocks, the dependence will not be broken: the earlier block will complete its traversal before the later block starts, preserving the ordering of the iterations. Hence, p_i and p_j must be in the same block. Further, for the dependence to be violated, we must have $(p_j, n_j) \prec (p_i, n_i)$ after applying point blocking.

We have three possible cases for the ordering of n_i and n_j:

$n_i \prec n_j$: In this case, n_i appears before n_j in the original program's traversal order. Recall that the block traverses the tree in the same order as the original points would have. Hence, the block will visit n_i before it visits n_j in the transformed code, preserving the dependence.

$n_i = n_j$: In this case, the points access the same location when they are at the same node in the tree. In the point blocked code, each point in a block executes its entire method body before moving on to the next point, so p_i performs its access before p_j, preserving the dependence.

$n_i \succ n_j$: In this case, n_j precedes n_i in the traversal order, so the block will visit n_j before it visits n_i, and (p_j, n_j) will occur before (p_i, n_i), violating the dependence.

Since we began by assuming the dependence must be violated, the third case must obtain. Hence, we have two iterations, (p_i, n_i) and (p_j, n_j), and two statements s_i and s_j such that: $s_i(p_i, n_i) \bowtie_e s_j(p_j, n_j)$, $p_i \prec p_j$ and $n_i \succ n_j$, violating the dependence test. □

DAG traversals Point blocking is applicable not only to traversals of trees, but to traversals of any recursive data structure, including DAGs and general graphs [15]. We note that the dependence test in Equation 2 is still valid for traversals of non-tree data structures. However, for DAGs and general graphs, the same node may be visited by a traversal more than once, so the $\succ$ relation between nodes in a traversal no longer obeys any sort of order. Because of the difficulty of determining the relation between two nodes in a DAG or graph traversal, if our analyses encounter a traversal of a data structure that cannot be proven to be a tree, we revert to applying Jo and Kulkarni's independence test for legality.

3.3 Simplified Dependence Tests

The dependence test of Equation 2 is difficult to apply. First, it can be hard to tell exactly when a statement might execute, due to complex, data-dependent control flow in the method body—not to mention that whether a particular iteration executes in the first place often depends on the structure of the tree, which is also input-dependent. Second, telling whether one node of the tree precedes another in the traversal order can also be tricky. We note, however, that we can simplify the dependence test in various ways while preserving soundness, as long as the resulting dependence test is at least as strong. In particular, the following dependence test is stronger than that of Equation 2:

$$\begin{aligned} &\forall\, p_i, p_j, n_i, n_j \,.\, (p_i \prec p_j) \rightarrow \\ &\qquad (\exists s_i, s_j \,.\, s_i(p_i, n_i) \bowtie_* s_j(p_j, n_j)) \rightarrow \\ &\qquad (n_i \preceq_a n_j) \end{aligned} \tag{3}$$

$$\begin{aligned}
v \in \mathit{Values} &::= \mathbb{Z} \qquad l \in \mathit{Locations} ::= \mathbb{L} \cup \textsf{null} \\
n \in \mathit{NodeRefs} &::= \textbf{root} \mid n_1 \mid n_2 \mid \ldots \\
\oplus &::= + \mid - \mid \times \mid \div \\
\odot &::= < \mid > \mid = \mid \neq \mid \geq \mid \leq \\
s \in \mathit{Stmts} &::= \textsf{skip} \mid \textsf{return} \mid s; s \mid c; \textsf{return} \\
&\mid \textsf{if}\ bexp\ \textsf{then}\ s\ \textsf{else}\ s \\
&\mid n := n \mid n := n.f_r \mid n.f_r := \textsf{null} \mid n.f_r := \textsf{alloc} \\
&\mid n.f_p := e \mid \textbf{point}.f_p := e \\
c \in \mathit{Calls} &::= \textbf{recurse}\,(\textbf{root}.f_r, \textbf{point}) \mid c; c \\
e \in \mathit{Exprs} &::= n.f_p \mid \textbf{point}.f_p \mid e \oplus e \mid v \\
bexp \in \mathit{BExprs} &::= n.f_r = \textsf{null} \mid n.f_r \neq \textsf{null} \mid e \odot e \\
p \in \mathit{Body} &::= s; \textsf{return}
\end{aligned}$$

Figure 3: Language for defining recursive tree traversals

where $\bowtie_*$ represents any interference test weaker than $\bowtie_e$, and $n_i \preceq_a n_j$ is the ancestry relationship, and is true iff $n_i = n_j$ or n_j is a descendant of n_i in the tree. Restated, the dependence test says that the transformation is safe when, for all iterations which are from two different points' traversals, if the two iterations interfere, the node where the earlier point's iteration occurs is an ancestor of the node where the later point's iteration occurs.

4. A Simple Language for Tree Traversals

To help formalize the discussion of our tree dependence analysis, we present a simple language for writing recursive tree traversal algorithms.[2] Because our analysis concerns itself with the behavior of the recursive method itself, rather than the code that invokes the method, the language is used to describe the body of a recursive method that traverses a tree, with arguments **root** and **point**, that define the *node* of the tree being visited and the *point* performing the traversal, respectively. Nodes are the objects that comprise the tree, while points are the objects that hold information local to each traversal. A frame program invokes the recursive method on the root of the tree for each of a set of points.

The points and nodes are structures, consisting of a number of fields. Tree node structures have one or more primitive fields, $f_p \in \mathbf{F}_p$ (holding values at each tree node), and one or more recursive fields, $f_r \in \mathbf{F}_r$ (references to their children in the tree), while point structures only have primitive fields.

4.1 Syntax and Assumptions

Figure 3 describes the syntax of recursive methods that traverse trees. Node references are local variables that can point to different nodes in the tree. There is a distinguished node reference, **root**, which names the reference passed in to the recursive method. Finally, there is a distinguished variable, **point**, that refers to the particular point structure passed in to the recursive method. For a given traversal of the tree, this point reference is fixed—the same reference is passed to all recursive invocations.

We note a few features that simplify reasoning about behavior. First, there are no loops in method bodies. Second, once a path through the method body reaches the recursive calls (c), it performs one or more recursive calls then returns, ensuring that all tree traversals are pre-order.

Note that the only means of manipulating the tree structure in a recursive method is by nullifying a subtree (by setting a recursive field to **null**), or by creating a fresh subtree (by setting a recursive field to point to a new tree node using **alloc**). Hence, if the traversal

[2] We use this specification language to cleanly present our analyses. Our implementation operates over Java programs whose operations are constrained to those supported by our specification language.

is called on a tree, we can be sure that after the traversal completes the resulting structure is still a tree. Proving that the initial structure is a tree is beyond the scope of this paper; shape analysis techniques can be used prior to our analyses to establish this fact. We assume that programs never dereference **null** fields. We also assume that programs initialize all fields of newly-allocated tree nodes before accessing them. We also assume that any local variable or node reference is only defined once along any path through the program.

Finally, we assume that the recursive method bodies are single callset (see Sections 2.2), ensuring a single, canonical traversal order. More formally, each straight-line sequence of recursive calls that occurs in the recursive method body induces a partial order on the recursive fields of **root**. If all of those partial orders are consistent with each other, the program is single callset.

Example programs Figure 5 shows how a quadtree traversal that occasionally updates a value at a node can be expressed in our simple language. Figure 6 shows how the BST insertion example from Figure 1 can be expressed.

4.2 Concrete Semantics

We define the semantics for programs written in our language in terms of the semantics of a particular tree traversal (*i.e.*, the semantics of a single iteration of the frame program's loop). A traversal operates over a heap, h, that contains a set of cells representing tree nodes. Each tree node's primitive fields map to values, while its recursive fields map to other heap locations or **null**. A subset of the tree nodes are linked together through their recursive fields to form the tree. The heap also contains a finite set of point structures.

During the execution of a traversal, a store σ maps references (including **root** and **point**) to heap locations.

$$\sigma : (\mathit{NodeRefs} \cup \mathbf{root} \cup \mathbf{point}) \rightarrow \mathbb{L}$$

The program state also contains a boolean return value, ρ, that tracks whether the method is supposed to return. Hence, the program state is a 3-tuple of the heap, the store, and ρ. The evaluation relation for statements and calls is: $\langle s,\ \sigma,\ h,\ \rho\rangle \rightarrow \langle\sigma',\ h',\ \rho'\rangle$ and the evaluation relation for expressions is: $\langle s,\ \sigma,\ h\rangle \rightarrow v$.

Figure 4 gives a subset of the concrete semantics for performing a traversal; the rules not shown follow the same pattern. The state at the beginning of a traversal is determined by the invocation of *recurse* by the frame program: $\langle p,\ \sigma[\mathbf{root} \mapsto \mathbf{tree}, \mathbf{point} \mapsto pt],\ h,\ \mathbf{F}\rangle$, where pt is a reference to the current point performing the traversal, and **root** starts out mapped to **tree**, the root of the tree structure (which resides in the heap). We assume that the tree structure has been initialized prior to beginning traversal. All other local variables are initialized to 0 or **null** as appropriate.

SKIP has standard semantics, leaving the store and heap untouched. RETURN changes the return flag to **T**. This flag is checked during statement sequencing (SEQ-RET and SEQ-CONT); if the first statement returns **T**, the second statement does not execute. IF-T has standard semantics, executing the true branch of the if statement; the semantics for the false branch are analogous. STORE-P stores the result into the appropriate point structure in the heap (looking up the heap location using σ).

Accessing tree nodes follows a similar pattern. DEF-N extracts the heap location pointed to by $n_2.f_r$, and maps n_1 to it. STORE-N dereferences n to update the primitive field of the appropriate tree node. ALLOC is similar to STORE-N, except that it updates the appropriate *recursive* field in the heap to point to a freshly-allocated tree node (with recursive fields initialized to **null** and primitive fields initialized to 0). The semantics for assigning null to a tree node's recursive field are similar.

Expressions have standard semantics. We show the rules for loading from **point** and references. Loading from **point** requires looking up which point structure is referenced in the store, then loading the appropriate field from the heap. Loading from a reference loops up the appropriate location in the store. Binary operations combine the results of their operands as expected.

```
root.v := root.v + 1;
if point.v = root.v
   return
else skip
if root.leaf = 1
   return
else skip
recurse (root.c1, point); recurse (root.c2, point);
recurse (root.c3, point); recurse (root.c4, point); return
```

Figure 5: Recursive method body for quadtree traversal

```
if root.v = −1
   root.v := point.v; return
else
   if root.v < point.v
      if root.l = null
         root.l = alloc; n1 := root.l; n1.v := −1
      else skip
      recurse (root.l, point); return
   else
      if root.r = null
         root.r = alloc; n1 := root.r; n1.v := −1
      else skip
      recurse (root.r, point); return
```

Figure 6: Recursive method body for BST insertion

The semantics of calls are relatively straightforward. The method body is re-executed with a new store, where **root** is remapped to the node the recursive call is invoked on and **point** retains the same mapping as the original store. Note that we do not remap any local variables; these variables will be re-initialized before being used. After the call returns, execution continues with the old store (thus returning to the old mapping for **root**), but the updated heap. Note, also, that the return flag of the call is always reset to **F**; if calls are sequenced, all calls execute, following the semantics of SEQ-CONT.

5. Path-Insensitive Dependence Analysis

Our first approach to dependence testing is a *path insensitive* analysis that assumes any statement in the method body might execute. This analysis proceeds in three steps:

1. Extracting the *rooted access paths* by associating every read and write to a field of a tree node in the method body with a field that can be reached through a series of field accesses starting from **root**.
2. Identifying *conflicting access paths* by determining whether, for two access expressions, at least one of which is performing a write, there exist two distinct nodes in the tree where if the first access path were rooted at the first node, and the second access path were rooted at the second, the two paths would refer to the same node.
3. Determining whether any conflicting access paths imply a possible dependence that precludes point blocking.

If step 3 yields no problematic accesses, then point blocking is legal. We now describe each of these steps in more detail.

5.1 Collecting Rooted Access Paths

First, reads and writes to tree nodes in the heap are transformed into reads or writes of *rooted access paths*. Access paths are elements of the regular set $\mathcal{A} = \mathbf{root}(.f_r)^*$ and *primitive* access paths are members of the set $\mathcal{A}_p = \mathbf{root}(.f_r)^*.(f_p \mid \iota)$. This lets us reason about the locations being read and written by the recursive method

$$\frac{l = \sigma(\mathbf{point}) \quad v = h(l.f_p)}{\langle \mathbf{point}.f_p,\ \sigma,\ h\rangle \to v}\ [\text{LOAD-P}] \qquad \frac{l = \sigma(n) \quad v = h(l.f_p)}{\langle n.f_p,\ \sigma,\ h\rangle \to v}\ [\text{LOAD-N}] \qquad \frac{\langle e_1,\ \sigma,\ h\rangle \to v_1 \quad \langle e_2,\ \sigma,\ h\rangle \to v_2 \quad v = v_1 \oplus v_2}{\langle e_1 \oplus e_2,\ \sigma,\ h\rangle \to v}\ [\text{BINOP}]$$

$$\langle \textbf{skip},\ \sigma,\ h,\ \mathsf{F}\rangle \to \langle \sigma,\ h,\ \mathsf{F}\rangle\ [\text{SKIP}] \qquad \langle \textbf{return},\ \sigma,\ h,\ \mathsf{F}\rangle \to \langle \sigma,\ h,\ \mathsf{T}\rangle\ [\text{RETURN}]$$

$$\frac{\langle s_1,\ \sigma,\ h,\ \mathsf{F}\rangle \to \langle \sigma',\ h',\ \mathsf{T}\rangle}{\langle s_1; s_2,\ \sigma,\ h,\ \mathsf{F}\rangle \to \langle \sigma',\ h',\ \mathsf{T}\rangle}\ [\text{SEQ-RET}] \qquad \frac{\langle s_1,\ \sigma,\ h,\ \mathsf{F}\rangle \to \langle \sigma',\ h',\ \mathsf{F}\rangle \qquad \langle s_2,\ \sigma',\ h',\ \mathsf{F}\rangle \to \langle \sigma'',\ h'',\ \rho\rangle}{\langle s_1; s_2,\ \sigma,\ h,\ \mathsf{F}\rangle \to \langle \sigma'',\ h'',\ \rho\rangle}\ [\text{SEQ-CONT}]$$

$$\frac{\langle e,\ \sigma,\ h\rangle \to v \qquad l = \sigma(\mathbf{point})}{\langle \mathbf{point}.f_p := e,\ \sigma,\ h,\ \mathsf{F}\rangle \to \langle \sigma,\ h[l.f_p \mapsto v],\ \mathsf{F}\rangle}\ [\text{STORE-P}] \qquad \frac{\langle e,\ \sigma,\ h\rangle \to v \qquad l = \sigma(n)}{\langle n.f_p := e,\ \sigma,\ h,\ \mathsf{F}\rangle \to \langle \sigma,\ h[l.f_p \mapsto v],\ \mathsf{F}\rangle}\ [\text{STORE-N}]$$

$$\frac{l_1 = \sigma(n_2) \qquad l_2 = h(l_1.f_r)}{\langle n_1 := n_2.f_r,\ \sigma,\ h,\ \mathsf{F}\rangle \to \langle \sigma[n_1 \mapsto l_2],\ h,\ \mathsf{F}\rangle}\ [\text{DEF-N}] \qquad \frac{l = \sigma(n)}{\langle n.f_r := \textbf{alloc},\ \sigma,\ h,\ \mathsf{F}\rangle \to \langle \sigma,\ h[l.f_r \mapsto \mathit{fresh}],\ \mathsf{F}\rangle}\ [\text{ALLOC}]$$

$$\frac{\langle \mathit{bexp},\ \sigma,\ h\rangle \to \mathsf{T} \qquad \langle s_1,\ \sigma,\ h,\ \mathsf{F}\rangle \to \langle \sigma',\ h',\ \rho'\rangle}{\langle \textbf{if}\ \mathit{bexp}\ \textbf{then}\ s_1\ \textbf{else}\ s_2,\ \sigma,\ h,\ \mathsf{F}\rangle \to \langle \sigma',\ h',\ \rho'\rangle}\ [\text{IF-T}] \qquad \frac{l = h(\sigma(\mathbf{root}).f_r) \qquad \langle p,\ \sigma[\mathbf{root} \mapsto l],\ h,\ \mathsf{F}\rangle \to \langle \sigma',\ h',\ \rho\rangle}{\langle \textbf{recurse}\ (\mathbf{root}.f_r, \mathbf{point}),\ \sigma,\ h,\ \mathsf{F}\rangle \to \langle \sigma,\ h',\ \mathsf{F}\rangle}\ [\text{CALL}]$$

Figure 4: Concrete semantics for traversal

relative to the current iteration (*i.e.*, the current values of **root** and **point**). The special field ι allows us to tell when the node itself is being read to or written from. We only consider accesses to tree nodes when looking for dependences. In our language, the point structures and local references accessed by each traversal are disjoint so cannot induce any cross-traversal dependences.

To collect the access paths, we define an abstract interpretation [6]. Intuitively, the abstract interpretation executes every path through the recursive method body, determining what (sets of) nodes each node reference can refer to, and associating with each read and write of a tree node field an access path starting from **root**. The analysis is loosely based on Wiedermann and Cook's approach to identifying paths traversed in object-relational databases [30].

The abstract store, $\hat{\sigma}$, maps primitive fields of **point** and primitive access paths to $\mathcal{P}(\mathbb{Z} \cup \perp)$, where $\perp$ represents unknown values; and maps **root** and node references to sets of access paths, $A \in \mathcal{P}(\mathcal{A})$. The program state consists of the abstract store, return flag (as in the concrete semantics), and two access path sets, $\pi_r, \pi_w \in \mathcal{P}(\mathcal{A}_p)$, which collect primitive access paths being read from and written to, respectively.

Intuitively, the abstract store soundly approximates both the concrete store and the concrete heap. Because **point** is fixed for each traversal, the specific location in the concrete heap that **point** refers to is irrelevant; the abstract store maintains the possible values of **point**'s fields directly. The concretization includes all possible **point** structures in the appropriate heap location. Node references in the concrete store always refer to nodes that are part of a subtree rooted at **root**. The abstract store captures these by mapping each node reference to a set of access paths; the concretization of those access paths are all of the cells in the heap that can be reached by following those access paths from **root**. Sets of primitive access paths are concretized similarly, with concretizations including all possible values for the primitive fields of the heap cells in the concretization.

The abstract semantics are given in Figure 7. The evaluation relation for statements and calls is $\langle s,\ \hat{\sigma},\ \pi_r,\ \pi_w,\ \rho\rangle \to \langle \hat{\sigma}',\ \pi_r',\ \pi_w',\ \rho'\rangle$, and the evaluation relation for expressions is $\langle e,\ \hat{\sigma}\rangle \to \langle \hat{v},\ \pi\rangle$. Note that expressions return a *set* of values, and can generate new access expressions. The sets of values arise because abstract stores are joined after conditionals, so node references can refer to multiple access paths, and primitive fields can take on multiple values. Expressions are always reads, so the evaluation relation generates only a single access path set. The initial abstract store maps all locals, primitive fields and primitive access paths to $\{\perp\}$, and maps **root** to $\{\mathbf{root}\}$ and everything else to $\emptyset$. The initial access path sets are $\pi_r = \{\mathbf{root}.\iota\}$ (recall that we assume that **root** is read in every iteration) and $\pi_w = \emptyset$.

Expressions (ALOAD-P, ALOAD-N) are handled as expected, with the only difference from the concrete semantics being that they return a *set* of values instead of just one, and that expressions that reference the tree (see ALOAD-N) can add accesses to the access set (note that those node references might refer to *multiple* access paths). Binary operations yield the result of applying the operation to all pairs of values from the two operands' value sets (with the operation yielding $\perp$ if one of the values is $\perp$).

We do not present the rules for **skip** and **return**, as they simply pass through the abstract store, heap and access path sets. The rules for sequencing of statements thread through the access path sets, setting the return flag and skipping over the execution of subsequent statements if necessary. Interestingly, calls (**recurse**) are handled much like **skip**. Even though a call reads an access path to make the recursive call, that read is instead associated with the beginning of the next iteration (see Section 3.2), and is captured by the initial access path set of $\mathbf{root}.\iota$.

ASTORE-N, which provides the semantics for $n.f_p := e$, shows an example of adding new access paths. After looking up the set of access paths that n is mapped to, for each such access path a, we add $a.f_p$ to the set of written access paths. The helper function `mapall` takes care of mapping each of the primitive access paths accessed by $n.f_p$ to the result of evaluating e. ADEF-N adds $a.f_r.\iota$ to the set of read access paths for all a that n_2 is mapped to.

AALLOC is interesting. It creates a new access path, indicating that $n.f_r.\iota$ has been written to. It only changes the store by setting the special primitive field $n.f_r.\iota$ to **alloc**. No other access paths are changed. In essence, our abstract semantics assume the tree structure itself already exists. Allocating a new node does not add a new node to the tree. Instead, it just writes to an existing node, as recorded by the access. The assumption that programs initialize fields before accessing them means that we do not have to worry about updating the values of any other fields.[3] A similar rule is used for **null**.

AIF runs both branches of the if statement, collecting the access paths from the boolean expression as well as both branches of the if statement. $\hat{\sigma}' \sqcup \hat{\sigma}''$ creates a new abstract store, where each variable or access path maps to the union of its mappings in $\hat{\sigma}'$ and $\hat{\sigma}''$. Note, too, that if both branches of the if statement call **return**, evaluating the if statement sets the return flag to true.

5.2 Identifying Conflicting Access Expressions

After collecting the accesses for the recursive method, the next step is to determine which accesses could result in dependences—two

[3] AALLOC introduces some inexactness to the set of accesses: if a new node is allocated for an access path, old node references that have the same access path will appear to access the new node as well. This does not affect soundness, as it can only introduce additional dependences.

$$\frac{\hat{v} = \hat{\sigma}(\mathbf{point}.f_p)}{\langle \mathbf{point}.f_p,\ \hat{\sigma}\rangle \to \langle \hat{v},\ \emptyset\rangle}\ \text{[ALOAD-P]} \quad \frac{A = \hat{\sigma}(n) \quad \hat{v} = \{\hat{\sigma}(a.f_p)\ \mid\ a \in A\}}{\langle n.f_p,\ \hat{\sigma}\rangle \to \langle \hat{v},\ \{a.f_p\ \mid\ a \in A\}\rangle}\ \text{[ALOAD-N]} \quad \frac{\langle e_1,\ \hat{\sigma}\rangle \to \langle \hat{v}_1,\ \pi_1\rangle \quad \langle e_2,\ \hat{\sigma}\rangle \to \langle \hat{v}_2,\ \pi_2\rangle \quad \hat{v} = \hat{v}_1 \hat{\oplus} \hat{v}_2}{\langle e_1 \oplus e_2,\ \hat{\sigma}\rangle \to \langle \hat{v},\ \pi_1 \cup \pi_2\rangle}\ \text{[ABINOP]}$$

$$\frac{\langle e,\ \hat{\sigma}\rangle \to \langle \hat{v},\ \pi_e\rangle \qquad A_1 = \hat{\sigma}(n) \qquad A_2 = \{a.f_p\ \mid\ a \in A_1\}}{\langle n.f_p := e,\ \sigma,\ \pi_r,\ \pi_w,\ \mathsf{F}\rangle \to \langle \hat{\sigma}[\mathsf{mapall}(A, f_p, \hat{v})],\ \pi_r \cup \pi_e,\ \pi_w \cup A_2,\ \mathsf{F}\rangle}\ \text{[ASTORE-N]}$$

$$\frac{A_1 = \sigma(n_2) \qquad A_2 = \{a.f_r\ \mid\ a \in A_1\}}{\langle n_1 := n_2.f_r,\ \hat{\sigma},\ \pi_r,\ \pi_w,\ \mathsf{F}\rangle \to \langle \hat{\sigma}[n_1 \mapsto A_2],\ \pi_r \cup \{a.\iota\ \mid\ a \in A_2\},\ \pi_w,\ \mathsf{F}\rangle}\ \text{[ADEF-N]}$$

$$\frac{A_1 = \sigma(n) \qquad A_2 = \{a.f_r\ \mid\ a \in A_1\}}{\langle n.f_r := \mathsf{alloc},\ \hat{\sigma},\ \pi_r,\ \pi_w,\ \mathsf{F}\rangle \to \langle \hat{\sigma},\ \pi_r,\ \pi_w \cup \{a.\iota\ \mid\ a \in A_2\},\ \mathsf{F}\rangle}\ \text{[AALLOC]}$$

$$\frac{\langle bexp,\ \hat{\sigma}\rangle \to \langle \hat{v},\ \pi_e\rangle \qquad \langle s_1,\ \hat{\sigma},\ \emptyset,\ \emptyset,\ \mathsf{F}\rangle \to \langle \hat{\sigma}',\ \pi_r',\ \pi_w',\ \rho'\rangle \qquad \langle s_2,\ \hat{\sigma},\ \emptyset,\ \emptyset,\ \mathsf{F}\rangle \to \langle \hat{\sigma}'',\ \pi_r'',\ \pi_w'',\ \rho''\rangle}{\langle \mathbf{if}\ bexp\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2,\ \hat{\sigma},\ \pi_r,\ \pi_w,\ \mathsf{F}\rangle \to \langle \hat{\sigma}' \sqcup \hat{\sigma}'',\ \pi_r \cup \pi_r' \cup \pi_r'' \cup \pi_e,\ \pi_w \cup \pi_w' \cup \pi_w'',\ \rho' \wedge \rho''\rangle}\ \text{[AIF]}$$

Figure 7: Abstract semantics to collect access expressions

accesses that touch the same location in the tree, with at least one of them a write.

Definition 1. *For a pair of accesses,* $\mathbf{root}.\alpha$ *and* $\mathbf{root}.\beta$*, we say that the two access paths* collide*—written* $\mathbf{root}.\alpha \sim \mathbf{root}.\beta$*—if there exists two nodes in a tree (of unbounded size),* n_1 *and* n_2 *such that* $n_1.\alpha$ *refers to the same location as* $n_2.\beta$*.*

This definition lends itself to a straightforward approach to finding access paths that collide. Suppose we consider the access path pair $\mathbf{root}.\alpha \in \pi_w$ and $\mathbf{root}.\beta \in (\pi_w \cup \pi_r)$. Without loss of generality, let α be the longer access path than β (*i.e.*, it contains at least as many field dereferences). We then have $\mathbf{root}.\alpha \sim \mathbf{root}.\beta$ iff β is a suffix of α.

If β is not a suffix of α, then, because the access paths traverse a tree, there is no way for the two to refer to the same field. Conversely, if β *is* a suffix of α, then let γ be a sequence of field accesses such that $\gamma.\beta = \alpha$. Note that γ's last field access must be a recursive field (if $\beta \neq \alpha$, otherwise $\gamma = \epsilon$). Then let n_1 be an arbitrary node in the tree (for example, the global root of the tree), and let n_2 be the node at $n_1.\gamma$. It is clear that $n_1.\alpha = n_2.\beta$.

If two access paths collide and one of them is a write, then there is a potential dependence between them. We can compute the set of such pairs, $S \subseteq \pi_w \times (\pi_w \cup \pi_r)$:

$$S = \{(a, b)\ \mid\ a \in \pi_w \wedge b \in (\pi_w \cup \pi_r) \wedge a \sim b\}$$

5.3 Applying the Dependence Test

After collecting the access paths, and identifying potential dependences, the final step is to determine whether the conflicting access paths preclude point blocking.

Note that the access paths in S are relative to $\mathbf{root}$, which is the index identifier for the traversal "loop" in the application. When iteration (p, n) executes a statement that reads from access path $\mathbf{root}.\alpha$, the field in the tree being read is $n.\alpha$. For each pair of conflicting access paths in S, $(\mathbf{root}.\alpha, \mathbf{root}.\beta)$[4], we compute γ as described previously. Let p_1 and p_2 be points such that $p_1 \prec p_2$. For all nodes n, during iteration $(p_1, n.\gamma)$, location $n.\gamma.\beta$ may be accessed by some statement s_1, and during iteration (p_2, n), location $n.\alpha$ may be accessed by some statement s_2. By the definition of conflicting accesses, we have $s_1(p_1, n.\gamma) \bowtie s_2(p_2, n)$.

By Equation 3, we see that for these potential dependences not to preclude point blocking, we must have $n.\gamma \preceq_a n$. We see that this can only be the case if $\gamma = \epsilon$. By verifying this condition for all pairs of conflicting access paths, we can determine whether point blocking is legal.

Soundness The key proof obligation to prove the soundness of this dependence analysis is to show that the set of access paths collected by the abstract interpretation allows us to find every s_i and s_j where there exist p_i, p_j, n_i, n_j such that $s_i(p_i, n_i) \bowtie_e s_j(p_j, n_j)$. If the set of statements we test for interference is a superset of these s_i and s_j, then the remainder of the dependence analysis (which ensures the proper ordering of p_i, p_j, n_i and n_j) soundly applies the dependence test from Equation 3. To show this, we show that if there are two statements that could interfere with each other in two specific iterations, we must have an pair of conflicting accesses that conflict in the same two iterations.

[4] Assume, without loss of generality, that β is a suffix of α.

Theorem 2. *If there exist* s_i *and* s_j *such that there exist* p_i, p_j, n_i, n_j *and* $s_i(p_i, n_i) \bowtie_e s_j(p_j, n_j)$*, there exists an access path pair* $(\mathbf{root}.\alpha, \mathbf{root}.\beta) \in S$ *such that* $n_i.\alpha = n_j.\beta$*.*

Proof sketch: Note that the only way for two statements to interfere in our language is if they access the same fields of a tree node. Note further that any tree node accessed by a recursive method body *must* be accessible from $\mathbf{root}$, and it can be accessed by only one path. Let us assume, without loss of generality, that s_i is a write and s_j is a read, and that the interference is through a primitive field access. Then s_i must be of the form $n_x.f_p := \ldots$ and s_j must contain an expression of the form $n_y.f_p$. By the antecedent, there must be some node m such that when $\mathbf{root}$ is mapped to n_i, $n_x = m$—and there must be exactly one access path $\mathbf{root}.\alpha = m$—and likewise, when $\mathbf{root}$ is mapped to n_j, there must be some access path $\mathbf{root}.\beta = m$. Hence, $n_i.\alpha = n_j.\beta$. By structural induction on the abstract semantics, upon encountering statement s_i, the abstract store will contain a mapping from n_x to $\mathbf{root}.\alpha$, adding the access $\mathbf{root}.\alpha.f_p$ to π_w; likewise, upon encountering s_j, $\mathbf{root}.\beta.f_p$ will be added to π_r. Because the abstract interpretation explores all paths, both accesses will be in the access path sets at the end of execution. Moreover, because both accesses refer to the same node in the tree, and each node in the tree can be accessed by only one path from the global root of the tree, either $\alpha.f_p$ will be a suffix of $\beta.f_p$ or vice versa, and the pair will be added to the set of conflicting accesses.

5.4 Examples

Quadtree traversal Running the abstract interpretation over the example from Figure 5 generates the following access paths: $\pi_w = \{\mathbf{root}.v\}$, $\pi_r = \{\mathbf{root}.\iota, \mathbf{root}.v, \mathbf{root}.leaf\}$ There is one pair of conflicting access paths: $(\mathbf{root}.v,\ \mathbf{root}.v)$. For two points, p_1 and p_2, with $p_1 \prec p_2$, iteration (p_1, n) writes to the same location that (p_2, n) does. For this pair, $\gamma = \epsilon$, so the dependence does not preclude point blocking. In particular, if p_1 and p_2 are in the same block, p_1 will perform its write before p_2 does, just as in the original, non-blocked code. Hence, despite the dependence between traversals, point blocking is legal for this code. Note that the simple dependence test of Jo and Kulkarni would have claimed that point blocking is illegal, as the traversals are not independent of each other.

$$v \in \mathbb{Z} \qquad b \in \{\mathsf{T}, \mathsf{F}\} \qquad a \in \mathcal{A} \qquad a_p \in \mathcal{A}_p$$
$$E = a_p \mid v \mid E \oplus E$$
$$P = E \odot E \mid a.\iota = \textsf{null} \mid a.\iota \neq \textsf{null}$$
$$F = b \mid P \mid F \wedge F \mid F \vee F$$

Figure 8: Logical fragment for path conditions

BST insertion Running the analysis over the BST insertion example from Figure 6 generates the following access paths:

$$\pi_w = \{\textbf{root}.v, \textbf{root}.l.\iota, \textbf{root}.l.v, \textbf{root}.r.\iota, \textbf{root}.r.v\}$$
$$\pi_r = \{\textbf{root}.\iota, \textbf{root}.v, \textbf{root}.l.\iota, \textbf{root}.r.\iota\}$$

Each access path in π_w conflicts with itself. But by the same analysis as in the quadtree example, these conflicts do not preclude point blocking: they all arise when different points are at the same node of the tree. However, the access paths $root.v \in \pi_r$ and $root.l.v \in \pi_w$ conflict with each other. Here, iteration $(p_1, n.l)$ reads from the same location that iteration (p_2, n) writes to. γ is l in this case, so the potential dependence precludes point blocking. However, we know that point blocking is legal for this code—our path-insensitive dependence analysis is too conservative. To develop a dependence analysis that correctly handles this code, we must also consider the *conditions* under which certain accesses happen.

6. Conditional Dependence Analysis

Even using the dependence test, the code in Figure 6 still exhibits a problematic dependence. The dependence test of the previous section assumes that all accesses in an iteration will happen. Consider two points p_1 and p_2 with $p_1 \prec p_2$, and the tree in Figure 2(a). When point p_1 is at node c, it reads from $c.v$ in line 1. That is the same field that point p_2 could write to at node b in line 6, when it writes to $\textbf{root}.l.v$.

However, reads and writes performed during traversals are not always unconditional in each iteration. It is often the case that if a traversal performs a particular access, other traversals *cannot* perform certain accesses: if iteration (p_1, c) reads from $c.v$, we see that iteration (p_1, b) must have established that $b.l \neq$ **null** (as that is the only way for **recurse** ($b.l$, **point**) to be called in line 8). Hence, when iteration (p_2, b) executes, it *will not* execute line 6, and the access that causes the problematic dependence does not happen.

This section describes how we augment the dependence analysis of the previous section to engage in this type of reasoning on conditions. The key insight is that we can determine the symbolic *path conditions* under which various accesses might occur, relative to arbitrary nodes in the tree. Given these conditions, we can prove that if the first of two potentially conflicting accesses occurs, the second cannot.

6.1 Attaching Conditions to Access Paths

First, we attach symbolic path conditions to each access path that can occur in a program. A path condition is some logical formula, $\phi \in (F \cup E)$ produced from the logical fragment given in Figure 8.

To track path conditions, we extend the abstract semantics of the previous section. First, we extend access paths to be a 3-tuple of an access path, a formula in the logic, and a flag that indicates whether the access path was a *strong* access. If an access path was generated by a variable dereference that only pointed to a single access path, the access path is strong, and is amenable to strong updates.

Expressions now yield formulae ($\Phi \in \mathcal{P}(F \cup E)$) in addition to sets of values (an expression can produce more than one conditional formula because variables accessed in an expression may map to more than one access path). Statements and expressions carry with them a *condition*, k, a predicate defining when statement might execute. The conditions capture a precondition that holds before a basic block executes. Hence, these conditions are updated when executing if statements. Figure 9 shows the relevant portion of the extended semantics. The evaluation relation for expressions is now $\langle e, \hat{\sigma}, k\rangle \to \langle \hat{v}, \pi_e, \Phi\rangle$ and the evaluation relation for statements is now $\langle s, \hat{\sigma}, \pi_r, \pi_w, k, \rho\rangle \to \langle \hat{\sigma}', \pi_r', \pi_w', k', \rho'\rangle$. The starting path condition for a program is T.

$$\begin{aligned}
\pi_w = \{&\textbf{root}.v\ [\textbf{root}.v = -1],\\
&\textbf{root}.l.\iota\ [\textbf{root}.v \neq -1 \wedge \textbf{root}.v < \textbf{point}.v \wedge \textbf{root}.l.\iota = \textsf{null}],\\
&\textbf{root}.l.v\ [\textbf{root}.v \neq -1 \wedge \textbf{root}.v < \textbf{point}.v \wedge \textbf{root}.l.\iota = \textsf{null}],\\
&\textbf{root}.r.\iota\ [\textbf{root}.v \neq -1 \wedge \textbf{root}.v \geq \textbf{point}.v \wedge \textbf{root}.r.\iota = \textsf{null}],\\
&\textbf{root}.r.v\ [\textbf{root}.v \neq -1 \wedge \textbf{root}.v \geq \textbf{point}.v \wedge \textbf{root}.r.\iota = \textsf{null}]\}\\
\pi_r = \{&\textbf{root}\ [\mathsf{T}], \textbf{root}.v\ [\mathsf{T}],\\
&\textbf{root}.l.\iota\ [\textbf{root}.v \neq -1 \wedge \textbf{root}.v < \textbf{point}.v],\\
&\textbf{root}.r.\iota\ [\textbf{root}.v \neq -1 \wedge \textbf{root}.v \geq \textbf{point}.v]\}
\end{aligned}$$

Figure 10: Conditional access paths in BST insertion

Expressions accessing fields generate atomic formulae as expected. When an expression generates an access path, the condition for the expression is attached to the access path. We also check the cardinality of the access path set in the store to determine whether the generated access path is a strong access. Comparison operations produce a new formula set from combining all pairs of formulae from its operands' formula sets (*e.g.*, if one operand has the formulae $\{\textbf{point}.x\}$ and the other has the formulae $\{1, 2\}$, then combining them with $\hat{=}$ produces the formula set $\{\textbf{point}.x = 1, \textbf{point}.y = 2\}$). We do not show rules for most statements; the only difference between these semantics and the semantics in Figure 7 is that when an access occurs, the statement's condition is associated with the access path. The strong tag is set, and a strong update performed on the abstract store, if the access path refers to exactly one node.

The key rules are for **if** statements. The formulae generated by the test condition are attached to the true and false branches of the **if** statement. If the test expression generates multiple formulae, the true branch is taken if *any* of the formulae are true, while the false branch is taken if *any* of the formulae are false; the conditions for the two branches are assembled appropriately. Joining together access paths ($\sqcup$) logically ors the conditions under which the access paths occur, and logically ands the strong tag.

The path condition after the **if** statement executes is subtle. It seems as though we should simply revert to the original condition, k, after control has re-converged. However, along one of the branches of the if statement, a write may have happened that invalidated part of the path condition. Consider **if root**.v $=$ 0 **then root**.v $:=$ 1 **else skip**. After the statement executes, we know that $\textbf{root}.v \neq 0 \vee \textbf{root}.v = 1$.

The helper function **munge**$(\hat{\sigma}, k, \pi_w)$ creates two formulae: k_1, which captures all possible values of access paths that were *definitely* written along the branch (determined by checking the strong tags); and k_2, which removes from k conditions that are invalidated by writes that *may* happen along the branch. The function returns $k_1 \wedge k_2$, which amounts to a postcondition for that branch of the **if** statement. The disjunction of the munged conditions from both branches of the **if** statement yields the *precondition* for the following statement. Note that if there are no writes along the branches, then the resulting path condition will again be k.

This treatment of **if** statements only occurs if the condition of the **if** statement accesses portions of the tree that have not yet been written (see the second premise of FIF1); otherwise, we pass no conditional information along branches of the **if** statement (FIF2). Figure 10 shows the results of running this analysis on our BST-insertion example (we elide the tag for strong accesses for brevity).

We can perform a very similar analysis (not shown, for lack of space) to determine under which conditions recursive calls are made. The only difference is that we also **munge** the path condition

$$\frac{\hat{v} = \hat{\sigma}(\mathbf{point}.f_p)}{\langle \mathbf{point}.f_p,\ \hat{\sigma},\ k\rangle \to \langle \hat{v},\ \emptyset,\ \{\mathbf{point}.f_p\}\rangle}\text{[FLOAD-P]} \qquad \frac{A = \hat{\sigma}(n) \quad \hat{v} = \{\hat{\sigma}(a.f_p) \mid a \in A\}}{\langle n.f_p,\ \hat{\sigma},\ k\rangle \to \langle \hat{v},\ \{a.f_p[k][|A| = 1] \mid a \in A\},\ \{a.f_p \mid a \in A\}\rangle}\text{[FLOAD-N]}$$

$$\frac{\langle e_1,\ \hat{\sigma},\ k\rangle \to \langle \hat{v_1},\ \pi_1,\ \Phi_1\rangle \quad \langle e_2,\ \hat{\sigma},\ k\rangle \to \langle \hat{v_2},\ \pi_2,\ \Phi_2\rangle}{\langle e_1 \odot e_2,\ \hat{\sigma},\ k\rangle \to \langle \{\bot\},\ \pi_1 \sqcup \pi_2,\ \Phi_1 \hat{\odot} \Phi_2\rangle}\text{[FCMPOP]}$$

$$\frac{\begin{array}{cc} \langle bexp,\ \hat{\sigma},\ k\rangle \to \langle \hat{v},\ \pi_e,\ \Phi\rangle & \forall a[k_*][b_*] \in \pi_w .\ \nexists a[k][*] \in \pi_e \\ \langle s_1,\ \hat{\sigma},\ \emptyset,\ \emptyset,\ k \wedge (\bigvee\{\phi \mid \phi \in \Phi\}),\ \mathsf{F}\rangle \to \langle \hat{\sigma}',\ \pi_r',\ \pi_w',\ k',\ \rho'\rangle & \langle s_2,\ \hat{\sigma},\ \emptyset,\ \emptyset,\ k \wedge (\bigvee\{\neg\phi \mid \phi \in \Phi\}),\ \mathsf{F}\rangle \to \langle \hat{\sigma}'',\ \pi_r'',\ \pi_w'',\ k'',\ \rho''\rangle \\ k_T = \mathsf{munge}(\hat{\sigma}', k', \pi_w') \wedge \neg\rho' & k_F = \mathsf{munge}(\hat{\sigma}'', k'', \pi_w'') \wedge \neg\rho'' \end{array}}{\langle \mathsf{if}\ bexp\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2,\ \hat{\sigma},\ \pi_r,\ \pi_w,\ k,\ \mathsf{F}\rangle \to \langle \hat{\sigma}' \sqcup \hat{\sigma}'',\ \pi_r \sqcup \pi_r' \sqcup \pi_r'' \sqcup \pi_e,\ \pi_w \sqcup \pi_w' \sqcup \pi_w'',\ k_T \vee k_F,\ \rho' \wedge \rho''\rangle}\text{[FIF1]}$$

$$\frac{\begin{array}{cc} \langle bexp,\ \hat{\sigma},\ k\rangle \to \langle \hat{v},\ \pi_e,\ \Phi\rangle & \text{FIF1 does not apply} \\ \langle s_1,\ \hat{\sigma},\ \emptyset,\ \emptyset,\ k,\ \mathsf{F}\rangle \to \langle \hat{\sigma}',\ \pi_r',\ \pi_w',\ k',\ \rho'\rangle & \langle s_2,\ \hat{\sigma},\ \emptyset,\ \emptyset,\ k,\ \mathsf{F}\rangle \to \langle \hat{\sigma}'',\ \pi_r'',\ \pi_w'',\ k'',\ \rho''\rangle \end{array}}{\langle \mathsf{if}\ bexp\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2,\ \hat{\sigma},\ \pi_r,\ \pi_w,\ k,\ \mathsf{F}\rangle \to \langle \hat{\sigma}' \sqcup \hat{\sigma}'',\ \pi_r \sqcup \pi_r' \sqcup \pi_r'' \sqcup \pi_e,\ \pi_w \sqcup \pi_w' \sqcup \pi_w'',\ k,\ \rho' \wedge \rho''\rangle}\text{[FIF2]}$$

Figure 9: Abstract semantics to collect conditional access expressions

prior to making the recursive call, to produce a precondition for the call. In essence, the condition we attach to the recursive call is a statement about the state of the tree when the call is made. For example, the condition for the recursive call in line 8 of Figure 6 is:

$$\mathbf{root}.v \neq -1 \wedge (\mathbf{root}.v < \mathbf{point}.v) \wedge$$
$$((\mathbf{root}.l.\iota = \mathsf{alloc} \wedge \mathbf{root}.l.v = -1) \vee \mathbf{root}.l.\iota \neq \mathsf{null})$$

6.2 Using Conditions to Disprove Dependences

Suppose we have a potential dependence between two accesses $(\mathbf{root}.\alpha[\phi_\alpha], \mathbf{root}.\beta[\phi_\beta])$ where $\alpha = \gamma.\beta$. The dependence that appears to preclude point blocking arises when $(p_1, n.\gamma)$ executes access path $\mathbf{root}.\beta$, and (p_2, n) executes access path $\mathbf{root}.\alpha$. The formulae ϕ_α and ϕ_β indicate the conditions under which the two accesses occur. If we can show that whenever ϕ_β is true during iteration $(p_1, n.\gamma)$, ϕ_α *will not* be true during iteration (p_2, n), then the dependence cannot arise. The procedure for doing this is as follows:

1. First, we construct a more precise condition for access $\mathbf{root}.\beta$. In particular, ϕ_β is a formula in terms of access paths rooted at **root**, which must be bound to the dynamic iteration instance. This is easily accomplished by substituting $n.\gamma$ for **root** to create ϕ'_β. We then substitute n for **root** to create ϕ'_α and query an SMT solver to determine ϕ'_β is incompatible with ϕ'_α. If so, we move to step 3.

2. ϕ'_β being compatible with ϕ'_α does not mean that both accesses will happen. ϕ'_β was computed with a starting path condition of T . To make the condition more precise, we *propagate* the conditions of the previous iteration down to $(p_1, n.\gamma)$. Define δ such that $\delta.f_r = \gamma$. If we substitute $n.\delta$ for the path conditions associated with all recursive calls **recurse** $(\mathbf{root}.f_r, \mathbf{point})$, we gain information about the state of the tree during iteration $(p_1, n.\delta)$, immediately before making a recursive call to start iteration $(p_1, n.\gamma)$. The disjunction of all such recursive conditions (call this ϕ_δ) is a sound approximation of the state of the tree before $(p_1, n.\gamma)$ executes. Essentially, we inline one instance of the recursive method. We then re-run the abstract interpretation with an initial condition of ϕ_δ, generating a stronger condition under which access $\mathbf{root}.\beta$ occurs.

   We repeat this "inlining" process, backing up one iteration at a time, until we reach iteration (p_1, n). We cannot inline beyond this point—n could be the global root of the tree, and hence there is no earlier iteration in the traversal. In practice, potentially-dependent iterations are nearby in the tree, so we need only inline one or two times.

   After performing this inlining, we have a much stronger path condition, ϕ'_β, for the problematic access. We can then query the SMT solver once again to determine whether the path conditions are incompatible. If they are not, then we declare this dependence a true conflict, and fail the overall dependence test.

3. If ϕ'_α is incompatible with ϕ'_β, we have determined that whatever computation p_1 performs during its traversal prevents p_2 from performing the access $\mathbf{root}.\alpha$. It is possible, however, for a traversal in between p_1 and p_2 to "reactivate" p_2's bad access. Thus, we must ensure that no other accesses can affect the path condition ϕ'_α that prevents p_2 from performing the bad access. We look for any access paths in π_w that collide with any access paths in ϕ'_α; these writes affect the path condition, and hence if some iteration performs the write, it may cause the bad access to occur. We use the same conditional dependence test to ensure that *those* accesses cannot happen. Note that any access path that appears in ϕ'_α must also appear in π_r. Hence, there are a bounded number of access paths to consider and the number of tests is finite.

6.3 Example

Consider the conflicting access paths $(\mathbf{root}.v[\mathsf{T}], \mathbf{root}.l.v[\mathbf{root}.v \neq -1 \wedge \mathbf{root}.v < \mathbf{point}.v \wedge \mathbf{root}.l.\iota = \mathsf{null}])$. These access paths preclude point blocking if iteration $(p_1, n.l)$ performs the first access and iteration (p_2, n) performs the second access. We substitute $n.l$ and n for the conditions to generate: $\phi'_\beta = \mathsf{T}$ and $\phi'_\alpha = n.v \neq -1 \wedge n.v < n.v \wedge n.l.\iota = \mathsf{null}$. These conditions are not incompatible with each other, so we "unroll" the recursive method by one iteration, passing the recursion condition from iteration (p_1, n) to $(p_1, n.l)$. The new ϕ'_β is:

$$n.v \neq -1 \wedge (n.v < n.v) \wedge ((n.l.\iota = \mathsf{alloc} \wedge n.l.v = -1) \vee n.l.\iota \neq \mathsf{null})$$

The refined condition under which iteration $(p_1, n.l)$ reads $n.l.v$ is clearly incompatible with the condition under which iteration (p_2, n) writes $n.l.v$—the latter requires that $n.l.\iota = \mathsf{null}$, while the former only happens when $n.l.\iota \neq \mathsf{null}$.

Finally, we must make sure that there is no intervening traversal that writes to $n.l.\iota$, possibly "re-activating" the write in iteration (p_2, n). We see that the only access path that writes to $n.l.\iota$ does so under the same condition as the write to $n.l.v$, and is therefore invalidated by the same argument. Repeating the process for all conflicting access paths, we discover that all pairs that might introduce a problematic dependence are incompatible with each other.

7. Generalization

The dependence test of Equation 2 is general: because it only refers to indices in the iteration space (points and nodes), it is not specific to traversals of trees. The strengthened form of the dependence test in Equation 3 makes two simplifications to make the test practical; each of these simplifications contributes to limiting the scope of our analysis. One simplification is that we do not verify that two statements definitely execute to produce a dependence; instead, we use the path-sensitive analysis of Section 6 to approximate which statements execute in a method body. To make this analysis effective, we limit our method bodies to loop-free execution. While more powerful analyses might be able to support more general method bodies, many recursive algorithms do not use statically-unbounded loops.

A more restrictive simplification is the replacement of the general ordering test for nodes ($n_i \succ n_j$) with a test for ancestry ($n_i \preceq_a n_j$). It is this restriction that limits our dependence test to operating over preorder tree traversals: it is only in such cases that a node's ancestors are guaranteed to be visited and fully processed before the node itself. Any attempt to generalize our dependence tests will require weakening this condition.

One generalization that is certainly worth exploring is relaxing the requirement of preorder traversals. For example, a postorder traversal merely flips the ancestry condition: descendants are processed before ancestors. By splitting method bodies into portions that execute before the recursive call and those that execute after the recursive call, we can even mix preorder and postorder execution in a single program. This relaxation can be especially effective for analyzing algorithms such as abstract syntax tree (AST) traversals in compilers; the legality of *traversal fusion*, which merges multiple AST passes, can be determined by a variant of our dependence test, as fusion is a limited case of point blocking.

There are several other generalizations to investigate. Our current specification language only allows localized changes to trees by adding or removing nodes. We could support more complex mutations of trees by defining mutating operations that generate more complex, or larger, set of access paths in our analyses to capture their effects on the tree. Such a generalization would also allow us to handle algorithms where entire subtrees are processed at once, rather than single nodes of a tree. We could support certain types of DAG traversals, as well: in many search algorithms, only the first visit to a node matters. Subsequent visits are either truncated, or some monotonic computation is performed. As a result, the DAG traversal functions as a (dynamic) tree traversal, allowing our analysis to be applied if the necessary preconditions hold.[5] These are all promising and exciting avenues for future work.

8. Implementation and Evaluation

Analysis implementation We implement our analysis in JastAdd [9], a compilation framework for Java.[6] The analysis analyzes recursive Java methods that are constrained to only use operations analogous to the operations in our specification language (Section 4); if a method does not obey those restrictions, we do not analyze it. We assume that either a shape analysis or a programmer annotation has established that the recursive data structure being traversed is a tree. The conditional analysis (Section 6) passes path conditions to the Z3 SMT solver [7], which checks whether they are compatible or not. The conditional analysis currently assumes that all writes used to compute post-conditions are strong (*i.e.*, in a single basic block, each write *definitely* happens), which is valid for the benchmarks we have studied.

Benchmarks We applied the dependence test of Equation 3 to five benchmarks, ranging from simple microbenchmarks to complex data-structure construction algorithms:

***ll*:** Repeatedly appending values to a linked list, with traversal starting from the head of the list.

***bst*:** Building a binary search tree, as in Figure 1.

***skew*:** Building a *skew-heap* [25].[7]

***bh*:** Building a Barnes-Hut octree.

***kdtree*:** Building a kd-tree using top-down insertion.

Our analysis is able to prove that the each of these benchmarks passes the dependence test, and hence can be soundly transformed using point blocking, as well as other optimizations; the following section describes the performance benefits of these transformations. Note that not only do all of these benchmarks modify the contents of the tree structure being traversed, they also *morph* the structure of the tree by adding additional nodes and edges.

Experimental configurations All experiments were run on a 32-core Intel Xeon system running at 2.7 GHz, with 256K of L2 cache per core, and 20MB of L3 cache shared among groups of 8 cores.

Analysis performance Table 1 summarizes the results of running our analysis on each benchmark. # access paths is the number of paths collected by our abstract interpretation. # interfering pairs is the number of access path pairs that interfere (*i.e.*, one is a suffix of the other). # conflicts is the number of pairs that require the conditional dependence analysis of Section 6 to rule out as problematic (the difference between this column and the previous column is the number of pairs that are ruled out by the path-insensitive analysis of Section 5). We also count how many Z3 calls are made for each benchmark, as benchmarks with more recursive calls require that more paths be checked to rule out conflicts. The upshot of these results is that for all five benchmarks, the simple independence test is not sufficient (some access paths interfere); moreover, the conditional analysis is *required* to verify the dependence test.

We measure both the overall analysis time, and the analysis time not including calls to Z3. Most of the benchmarks are analyzed very quickly. We note that ***bh*** takes quite a bit longer than the other benchmarks, due both to the larger number of access paths and to the 8 recursive calls in the method body, which leads to a commensurate increase in the number of Z3 calls.

Transformation evaluation After proving that the benchmarks pass the dependence test, we applied three different transformations whose legality is established by the dependence test:

1. *Point blocking*, described in detail in Section 2.2. The baseline code for point-blocking is written in Java (and is the same code analyzed by the analysis framework described above). Point blocking uses a block size equal to the input size.
2. *Traversal splicing* [16]. In contrast to point blocking, traversal splicing tiles the "tree loop" instead of the point loop. The original version of traversal splicing reorders the point loop during execution, and hence is not amenable to the dependence test that we develop in this work. However, for benchmarks where a point only visits one child of any node, traversal splicing performs no reordering, and hence is legal whenever the dependence test of Equation 3 holds. For infrastructural reasons, the baseline for traversal splicing is written in C++: we analyzed the Java version of the benchmarks to prove the transformations' legality, then ported the benchmarks to C++.
3. *Parallelization*. It is well-known that top-down tree building algorithms can be parallelized by recursively building left and right subtrees in parallel. We design a transformation that derives the parallel implementation from the sequential version of any traversal code where each point only visits one child of a node: we apply point blocking to the code, and can then simply run each of the left and right recursive calls (e.g., the two recursive calls in Figure 1(b)) in parallel. The resulting parallel implementation not only requires no locks, it is also guaranteed to produce the same tree as the original sequential code. We used Cilk+ [11] for parallelism. We ran the parallel code using

[5] Extending the analysis beyond trees will require a different interference test—using access path prefixes assumes tree-ness.

[6] Our analysis framework has been certified by the artifact evaluation committee, and can be downloaded, along with our benchmarks, at `https://bitbucket.org/plcl/tree-dependence-analysis`.

[7] We modify the algorithm slightly to fit our language restrictions.

Table 1: Analysis results. LoC is lines of code in the analyzed method. Runtimes are in seconds, and include 95% confidence intervals.

Benchmark	LoC	# access paths	# interfering pairs	# conflicts	# Z3 calls	Runtime (no Z3)	Runtime (total)
ll	9	5	5	1	1	0.7914 ± .0945	0.8174 ± 0.0959
bst	21	12	20	8	16	0.8800 ± 0.136	1.220 ± 0.154
skew	28	21	41	16	32	0.9527 ± 0.0430	1.687 ± 0.0498
kdtree	145	145	901	510	3060	23.94 ± 0.462	109.0 ± 0.481
bh	829	780	40248	3448	27584	280.3 ± 6.78	1432 ± 16.5

Table 2: Speedups of transformed versions over the baselines specified in the text (95% confidence intervals in parentheses).

Bench.	Blocking	Splicing	Parallelization
ll	1.42 (1.39, 1.45)	N/A	N/A
bst	2.59 (2.52, 2.65)	2.00 (1.87, 2.13)	1.54 (1.53, 1.56)
skew	1.59 (1.53, 1.66)	0.86 (0.85, 0.86)	0.76 (0.73, 0.79)
kdtree	1.80 (1.75, 1.85)	2.65 (2.64, 2.66)	2.07 (2.01, 2.12)
bh	1.17 (1.15, 1.18)	1.28 (1.27, 1.29)	2.67 (2.58, 2.75)

4 threads, and compared to a baseline of the Cilk+ code running on a single thread.

For ***ll***, we insert 60,000 values; to avoid stack overflow, we perform tail-call optimization on the *transformed* code. For each of the other benchmarks, we build the trees using 10 million points/-values. The splicing and parallelization transformations are only applied to the four tree-based benchmarks. Table 2 presents the results. Each experiment is run 15 times, and the table shows the average speedups, with 95% confidence intervals.

Our transformations are able to achieve substantial speedups on most of the benchmarks. The exception is ***skew***, which has slowdowns for splicing or parallelization, despite good speedup for point blocking. In the case of splicing, we believe this is because the overhead of splicing far outweighs its locality benefits—point blocking is a lighter-weight transformation. Parallelization has no speedup for ***skew*** due to low available parallelism (there is relatively more work to be done at the root node, which must be done sequentially) and Cilk overheads (we observe a 1.13× speedup when using two cores, which dissipates when using four cores).

Each of these transformations is enabled by our dependence test, and, moreover, *would not* have been proven legal by prior dependence tests, including Jo and Kulkarni's [15]. Our goal with these experiments is not to evaluate these transformations against each other; indeed, these transformations are not a contribution of this work. Instead, our aim is to show that extending these transformations to a wider class of kernels through our dependence test and tree dependence analysis is beneficial. Note, for example, that by applying our dependence analysis to ***bh***, almost the entirety of the application—the two major kernels, tree building and tree traversal, comprise 99% of its runtime—is now amenable to point blocking and traversal splicing.

9. Related Work

Many analyses have taken access-path based approaches to reasoning about the behavior of programs. We adopt the notion of attaching conditions to access paths to facilitate deeper reasoning from Wiedermann and Cook [30]. Their domain (reasoning about object-relational database programs) is substantially different than ours, and they only consider read accesses. Our notions of interference of statements and collisions of access paths are similar to those defined by Larus and Hilfinger [18]. Hummel *et al.* also use the interference of access paths to identify dependences [14]. Ghiya *et al.* use shape analysis to determine whether loop iterations can be parallelized [13]. Rugina and Rinard use symbolic constraints to determine possible interference between pointers [23]. Other shape analyses can prove interesting properties of structures, but do not provide enough information to reason about dependences [8, 12, 24]. To our knowledge, ours is the first analysis to reason about the *structure* of dependences with respect to recursion (rather than just the existence of dependences).

In the context of reasoning about tree and graph programs, Andrusky *et al.* reason about the kinds of traversals performed over trees (breadth-first vs. depth-first) [3]; their approach does not reason about dependences. Rinard and Diniz use *commutativity analysis* to prove that certain dependences do not preclude parallelization of graph traversals [22]; we instead reason about structural properties of non-commutative dependences (as in BST insertion). Zumbusch reasons about dependences to parallelize individual tree traversals, but does not consider the relationship between multiple traversals [33]. Madhusudan *et al.* develop a logic for reasoning about recursive invariants on trees [19]; in contrast, we reason about the dynamic behavior of a program, and, in particular, the behavior of multiple operations (traversals) over trees.

Most dependence analysis frameworks ([1, 4, 5, 10, 17, 20, 31, 32], among numerous others) operate over nested loops with affine loop bounds that manipulate arrays using affine subscripts. More recent work has attempted to generalize this model to handlenon-affine loop bounds and subscripts using symbolic expressions [21, 28, 29], but they still confine themselves to loop-based programs; these approaches also require a run-time component to evaluate the symbolic expressions. Similarly, there has been substantial work in applying locality transformations to sparse-matrix programs using hybrid compile-time/run-time approaches [26, 27]. None of these approaches deal with recursion and pointer-based structures, however. In addition, many of these approaches deal with perfectly- or imperfectly-nested loops, but do not consider further control flow within loop bodies; our path condition–based analysis is able to account for control flow in ruling out potential dependences.

10. Conclusions

This paper presents techniques for analyzing dependences in programs that recursively traverse trees. We developed an accurate dependence test that identified only those dependences that preclude point blocking. Through a conditional tree dependence analysis, we prove the legality of point blocking and other transformations for a wide range of programs, including tree building codes.

Through multiple decades of compiler research sophisticated dependence analysis frameworks like the unimodular and polyhedral frameworks were developed to apply transformations like loop tiling to array programs in the face of complex dependences. Despite these decades of research, similar analyses for pointer-based programs have been an elusive target. This paper presents the first dependence analysis toolkit that can prove the legality of analogous "loop" transformations over pointer-based data structures.

Acknowledgments

The authors would like to thank the anonymous referees for their comments and suggestions. This work was supported in part by an NSF CAREER award (CCF-1150013) and a DOE Early Career award (DE-SC0010295).

References

[1] J. R. Allen and K. Kennedy. Automatic Loop Interchange. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, pages 233–246, 1984.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2001.

[3] K. Andrusky, S. Curial, and J. N. Amaral. Tree-traversal orientation analysis. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*, pages 220–234, 2006.

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

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–113, 2008.

[6] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[7] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] A. Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 230–241, New York, NY, USA, 1994. ACM.

[9] T. Ekman and G. Hedin. The JastAdd Extensible Java Compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[10] P. Feautrier. Some Efficient Solutions to the Affine Scheduling Problem: I. One-dimensional Time. *Int. J. Parallel Program.*, 21:313–348, October 1992.

[11] M. Frigo, C. E. Leiserson, and K. 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.

[12] 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 *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 1–15, New York, NY, USA, 1996. ACM.

[13] 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.

[14] J. Hummel, L. J. Hendren, and A. Nicolau. A General Data Dependence Test for Dynamic, Pointer-based Data Structures. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 218–229, New York, NY, USA, 1994. ACM.

[15] Y. Jo and M. Kulkarni. Enhancing Locality for Recursive Traversals of Recursive Structures. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 463–482, 2011.

[16] Y. Jo and M. 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*, pages 355–374, 2012.

[17] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.

[18] J. R. Larus and P. N. Hilfinger. Detecting Conflicts Between Structure Accesses. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 24–31, New York, NY, USA, 1988. ACM.

[19] P. Madhusudan, X. Qiu, and A. Stefanescu. Recursive Proofs for Inductive Tree Data-structures. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 123–136, New York, NY, USA, 2012. ACM.

[20] W. Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM.

[21] W. Pugh and D. Wonnacott. Non-Linear Array Dependence Analysis. *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 1–14, 1996.

[22] M. C. Rinard and P. C. Diniz. Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers. *ACM Trans. Program. Lang. Syst.*, 19(6):942–991, Nov. 1997.

[23] R. Rugina and M. C. Rinard. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. *ACM Trans. Program. Lang. Syst.*, 27(2):185–235, Mar. 2005.

[24] M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, May 2002.

[25] D. D. Sleator and R. E. Tarjan. Self Adjusting Heaps. *SIAM J. Comput.*, 15(1):52–69, Feb. 1986.

[26] M. M. Strout, L. Carter, and J. Ferrante. Compile-time Composition of Run-time Data and Iteration Reorderings. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 91–102, New York, NY, USA, 2003. ACM.

[27] M. M. Strout, F. Luporini, C. D. Krieger, C. Bertolli, G.-T. Bercea, C. Olschanowsky, J. Ramanujam, and P. H. J. Kelly. Generalizing run-time tiling with the loop chain abstraction. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 1136–1145, Washington, DC, USA, 2014. IEEE Computer Society.

[28] R. A. van Engelen, J. Birch, Y. Shou, B. Walsh, and K. A. Gallivan. A Unified Framework for Nonlinear Dependence Testing and Symbolic Analysis. In *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS '04, pages 106–115, New York, NY, USA, 2004. ACM.

[29] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout. Non-affine Extensions to Polyhedral Code Generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 185:185–185:194, New York, NY, USA, 2014. ACM.

[30] B. Wiedermann and W. R. Cook. Extracting Queries by Static Analysis of Transparent Persistence. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 199–210, New York, NY, USA, 2007. ACM.

[31] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 30–44, New York, NY, USA, 1991. ACM.

[32] M. Wolfe. More Iteration Space Tiling. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, pages 655–664, 1989.

[33] G. Zumbusch. Data Dependence Analysis for the Parallelization of Numerical Tree Codes. In B. Kågström, E. Elmroth, J. Dongarra, and J. Waśniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 890–899. Springer Berlin Heidelberg, 2007.