Fall 2013

Automatically Optimizing Tree Traversal Algorithms

Youngjoon Jo
Purdue University

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Youngjoon Jo

Entitled
Automatically Optimizing Tree Traversal Algorithms

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

MILIND KULKARNI
Chair

ANAND RAGHUNATHAN

MITHUNA S. THOTTETHODI

SAMUEL P. MIDKIFF

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): MILIND KULKARNI

Approved by: M. R. Melloch 08-27-2013
Head of the Graduate Program Date

AUTOMATICALLY OPTIMIZING TREE TRAVERSAL ALGORITHMS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Youngjoon Jo

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2013

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

Foremost, I have the utmost gratitude to my fabulous advisor Milind Kulkarni, who has shepherded, mentored and minted me into a PhD. I had the fortune of starting research with Milind in my first year, and my journey has been both fulfilling and enlightening since. Every time I got stuck on a problem, Milind was there to provide insights to transform mere thoughts into concrete foundations with which to overcome roadblocks. Milind has always been supportive of letting me pursue my own ideas at my own pace, and the need to cool down from time to time. I believe having a great advisor with whom you can build rapport is the most important part of the PhD process, and I can truly ask for no better on that.

I am indebted to many other mentors. I thank my advisory committee, Sam Midkiff, Anand Raghunathan and Mithuna Thottethodi for attending my examinations, and providing feedback to improve my dissertation and general grains of wisdom. A joint research project with Mithuna, and his courses on algorithms and architecture gave me skills to conduct my research. I thank Ilya Kirnos, Sean Foy and Deepak Jindal for mentoring me during my internship on the Gmail team at Google. That internship was an eye-opener for me in the scale and atmosphere of industry, and my experiences there were critical in landing my upcoming job at Google. I have sincere gratitude to Todd Mytkowicz, Aman Kansal and Kathryn McKinley for mentoring me during my internship at Microsoft Research. They taught me how to approach completely new problems at various angles and the skills in collaborating with diverse people. Todd went out of his way to make my time at MSR exciting, and provided valuable discussions on the vectorization work of this dissertation. I thank Hyuk-Jae Lee for mentoring me as an undergrad researcher in his group at Seoul National University, and guiding me towards graduate studies at Purdue.

Purdue is a bucolic setting, and life can be dull at times if not for many good friends who were there to cheer me up. I cannot possibly hope to express my gratitude to all my friends, and must instead list a few and hope that others not recognized understand that the

omission is one of forgetfulness and oversight, not of intent. I thank Minwoong, Yukeun and Junhyung for being good housemates and great friends. I thank Hyundok, Sunghoon, Junil, Jacques, Jeff, Heejin, Hyokun, Woosuk, Kangwoo, Woori, Inkyung, Jason, Jeremy, Ravi, Nitin, Kanad, Ahmed and Keith for the occasional beer, racquetball, Starcraft and other fun to keep me happy and sane. I thank my groupmates, Mike, Jad, Hasan and Nabeel for their friendship, and being around to provide a productive and enjoyable work environment. Mike provided insights on my research and was fun to collaborate with and be around. I thank all my friends in Korea and the U.S. for welcoming me when I needed to chill.

Last but not least, I must express gratitude to my family for their support throughout my life. My parents and brother always provided an ear to talk to on the phone, and a place to unwind on my trips back home. My fiancee, Jiyoung, has been my source of emotional support and is a great blessing to me.

This research was supported in part by an NSF CAREER Award (CCF-1150013) and grants from the Purdue Research Foundation and Intel.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Jo, Youngjoon Ph.D., Purdue University, December 2013. Automatically Optimizing Tree Traversal Algorithms. Major Professor: Milind Kulkarni.

Many domains in computer science, from data-mining to graphics to computational astrophysics, focus heavily on *irregular applications*. In contrast to *regular* applications, which operate over dense matrices and arrays, irregular programs manipulate and traverse complex data structures like trees and graphs. As irregular applications operate on ever larger datasets, their performance suffers from poor locality and parallelism. Programmers are burdened with the arduous task of manually tuning such applications for better performance. Generally applicable techniques to optimize irregular applications are highly desired, yet scarce.

In this dissertation, we argue that, for an important subset of irregular programs which arises in many domains, namely, tree traversal algorithms like Barnes-Hut, nearest neighbor and ray tracing, there exist general techniques to enhance performance. We investigate two sources of performance improvement: locality enhancement and vectorization. Furthermore we demonstrate that these techniques can be automatically applied by an optimizing compiler, relieving programmers of manual, error-prone, application-specific effort.

Achieving high performance in many applications requires achieving good locality of reference. We propose two novel transformations called point blocking and traversal splicing, inspired by the classic tiling loop transformation, and show that it can substantially enhance temporal locality in tree traversals. We then present a transformation framework called TREESPLICER, that automatically applies these transformations, and uses autotuning techniques to determine appropriate parameters for the transformations. For six benchmark algorithms, we show that a combination of point blocking and traversal splicing can

deliver single-thread speedups of up to 8.71 (geometric mean: 2.48), just from better locality.

Modern commodity processors support SIMD instructions, and using these instructions to process multiple traversals at once has the potential to provide substantial performance improvements. Unfortunately tree algorithms often feature highly diverging traversals which inhibit efficient SIMD utilization, to the point that other, less profitable sources of vectorization must be exploited instead. We propose a dynamic *reordering* of traversals based on previous behavior, based on the insight that traversals which have behaved similarly so far are likely to behave similarly in the future, and show that this reordering can dramatically improve the SIMD utilization of diverging traversals, close to ideal utilization. We present a transformation framework, SIMTREE, which facilitates vectorization of tree algorithms, and demonstrate speedups of up to 6.59 (geometric mean: 2.78). Furthermore our techniques can effectively SIMDize algorithms that prior, manual vectorization attempts could not.

1. INTRODUCTION

1.1 Motivation

Avatar was an impressive film released in 2009 which currently holds the worldwide box office sales record[1]. But Avatar was also a ray traced film which took on average 30-50 hours to render each frame [1]. For 24 frames per second, this would be 8 million hours to render 2 hours of film on a single processor. The lead visual effects company for Avatar used a $10{,}000$ square foot server farm with $35{,}000$ cores for rendering.

Computer graphics rendering for Avatar is one example of the many compute intensive tasks that are crunched in massive data centers around the world today. Google employs more than a million cores in its data centers worldwide to continuously index the evolving web, and deliver search results with milliseconds of latency (among many other tasks). Many of these data center workloads from data-mining to graphics to computational astrophysics, focus heavily on *irregular applications*. In contrast to *regular* applications, which operate over dense matrices and arrays, irregular programs manipulate and traverse complex data structures like trees and graphs. Ray tracing builds a tree over the objects in the scene and repeatedly traverses the tree to determine the color of each pixel. Nearest neighbor algorithms build a tree over a dataset, and traverse the tree to find the nearest neighbor of each data point.

Such irregular programs perform complex operations on complex data structures, and hence writing high-performance implementations is difficult. This difficulty is exacerbated by increasing input sizes, and the poor locality of reference that follows from it. Poor locality in turn results in poor parallelism, and both result in poor performance. In contrast to the decades of work on developing automatic compiler optimizations to enhance the performance of regular programs, most attempts at optimizing irregular programs have

[1]The record was previously held by Titanic released in 1997, also directed by James Cameron.

been ad-hoc and application specific. Programmers are burdened with the arduous task of manually tuning such applications for better performance. Generally applicable techniques to optimize irregular applications are highly desired, yet scarce.

The chief obstacle to identifying and applying performance-enhancing transformations on irregular applications is the apparent lack of principles that unify them. This apparent lack of structure in irregular programs can be misleading. While the particular set of concrete memory accesses may exhibit little regularity, at an abstract level there are organizing principles governing these accesses, such as the topology of the irregular data structure, or the nature of operations on that data structure. Recent work by Pingali *et al.* has suggested that there may, indeed, be significant structure latent in irregular applications [2]. Can this structure be exploited to transform irregular applications so as to enhance locality and facilitate vectorization?

One important class of irregular applications is *traversal codes*, applications that perform repeated traversals of irregular data structures. Examples include well-known scientific algorithms such as Barnes-Hut [3], data mining algorithms such as point correlation and nearest neighbor [4], and graphics algorithms like ray tracing with bounding volume hierarchies [5]. These algorithms feature repeated traversals of highly irregular trees, with unpredictable application- and input-dependent traversal sizes, shapes and orders. We investigate two sources of performance improvement for tree algorithms: locality enhancement [6, 7] and vectorization [8].

Exploiting locality in these algorithms is critical because their performance is dominated by memory-access time, and careless accesses to irregular data structures are likely to result in cache misses. Any technique that can turn a substantial portion of those misses into hits has the potential to dramatically improve performance. Once locality is improved we can turn our attention to parallelization. We look into vectorization techniques to take advantage of SIMD instructions available on commodity processors.

1.2 Locality in tree algorithms

While there has been much work on automatic techniques for improving locality in *regular* programs, which operate over dense matrices and arrays [9], there has been comparatively little work on general techniques for improving locality in *irregular* programs, which operate over pointer-based data structures such as trees and graphs. There has been various work to enhance the *spatial* locality of pointer based structures, either at allocation time or through garbage collection, some of which are *automatic* [10–15]. On the other hand, attempts at enhancing *temporal* locality has focused on *ad hoc* techniques that leverage application semantics [16–24], or work well for sparse-matrix style algorithms [25–27], but not the tree- and graph-based algorithms that proliferate in domains like data mining and graphics.

Prior work focusing on temporal locality in tree algorihtms has proposed changing the order in which points (*i.e.* the entities that traverse the tree) are processed [16, 17, 19, 22]. By "sorting" the points such that points processed consecutively have similar traversals, locality can be enhanced. Unfortunately, performing point sorting requires analyzing the points prior to the traversals to rearrange them effectively. Determining an efficient way of performing this *a priori* sort requires an understanding of the semantics of the algorithm and hence highly application-specific techniques. Indeed, for some algorithms, the traversals are so complex that it is unclear how to do an *a priori* sort of the points to maximize traversal overlap, even when armed with semantic knowledge.

We develop an abstract model of tree traversal codes that analogizes them to doubly-nested loops as seen in regular algorithms like vector outer product. The outer loop is a loop of *points* that must traverse the tree, while the inner loop is a loop over the *nodes* that make up the traversal, irrespective of their position in the tree. Using this model, we propose two novel transformations akin to loop tiling in regular loop programs.

The first is *point blocking* [6], which essentially "tiles" the point loop: rather than performing a single point's entire traversal before moving on to the next point, a group of points are placed into a block, and the block traverses the tree, with each point in the block

interacting with the necessary portions of the tree. Point blocking can deliver substantial locality and performance improvements for large traversals. Unfortunately, point blocking's performance is sensitive to the order in which points are processed; achieving the best possible performance from point blocking requires that the point sorting optimization be performed. As a result, point blocking is not a truly automatic, application-agnostic technique, depending instead on point sorting.

The second transformation we propose is *traversal splicing* [7]. Much as point blocking tiles the point loop, traversal splicing tiles the *traversal* loop: each point's traversal is divided into a number of partial traversals, and we perform a partial traversal for all points before moving on to the next partial traversal for any point. There are two key advantages to traversal splicing. First, the performance of splicing is largely independent of the order of points, decoupling its behavior from application-specific sorting transformations. Second, the order in which partial traversals are performed can be changed during execution. In particular, *as points traverse the tree*, we can use their traversal history as a predictor of their remaining traversal patterns, and group similar points together. In essence, traversal splicing *sorts the points on the fly* but *without any application-specific optimization*.

We develop TREESPLICER[2], a compiler framework that automatically identifies regions of programs where data reuse implies that point blocking and traversal splicing might be successfully applied. In regular programs, data reuse often arises in nested loops that manipulate arrays and matrices, and can be readily identified. In irregular programs, in contrast, data reuse is often masked by pointer-manipulation operations. TREESPLICER identifies code where point blocking might be performed by looking for *recursive traversals of recursive structures*. If point blocking and traversal splicing is legal for such a traversal, TREESPLICER automatically performs the transformation.

Point blocking and traversal splicing, like loop tiling, requires that optimization parameters be carefully tuned to match both the application and the architecture. *Autotuning* has emerged as a popular approach to parameter selection as it can select optimization parameters for a particular execution scenario without programmer intervention [28–30], a

[2] https://engineering.purdue.edu/plcl/treesplicer/

necessity for any automated transformation framework. Because irregular programs are highly input-dependent, TREESPLICER uses run-time profiling to guide its selection of parameters for our transformations.

We evaluate TREESPLICER on six benchmark algorithms, and show that our transformations deliver (single-thread) speedups of up to 8.71 (geometric mean: 2.48) when compared to straightforward implementations of these algorithms. We present comprehensive results of all combinations of point sorting, point blocking and/or traversal splicing, and find that point blocking with traversal splicing, which can be applied fully automatically, is competitive with hand optimized implementations which exploit semantics based point sorting.

1.3 Vectorizing tree algorithms

An effective way to improve the performance of a program run on modern commodity architectures is to *vectorize* the application by exploiting Single-Instruction Multiple-Data (SIMD) instructions. By using SIMD instructions, multiple scalar operations can be replaced with a single vector operation that acts on multiple pieces of data at once. Because SIMD extensions typically require relatively little extra hardware, exploiting SIMD parallelism is effectively "free" from a power-consumption standpoint, making SIMD an attractive approach to improving performance per watt on modern architectures.

While modern vectorizing compilers (*e.g.*, Intel's icc, IBM's xlc, and GCC) can target SIMD architectures, they primarily rely on identifying simple loop-based control structures with predictable access patterns to generate effective SIMD code [31]. While these patterns arise frequently in *regular* programs, which operate on dense matrices and arrays, they are far less common when dealing with *irregular* programs, which operate over pointer-based data structures. These programs are characterized by complex, data-dependent access patterns, hence complicating vectorization, which requires regularizing computation so that multiple pieces of data are simultaneously operated on by a single instruction. As a result, vectorization of such pointer-based programs has been the province of careful, handwrit-

ten, application-specific implementations [5, 32–38]. Generally-applicable approaches to "SIMDization" of irregular programs are scarce.

In tree algorithms a single tree is traversed multiple times by multiple points. Hence, if multiple points that visit the same tree node can be processed simultaneously, there will be several common operations that differ only in the (point-specific) data they operate on, a ripe opportunity for SIMD acceleration. In essence, a general approach to exploiting SIMD requires carefully scheduling traversals to ensure that multiple points are at a single tree node at once, and can be processed in a SIMD manner.

Scheduling traversals so that multiple points are visiting a node of the tree simultaneously is non-trivial. A natural scheduling approach is to group four points with similar traversals into a single *packet*. This packet then traverses the tree in lockstep, allowing each of the points in the packet to interact with the tree nodes the packet visits [5, 32]. However, there are two drawbacks to packetized traversals. First, writing a packet-based traversal requires making substantial changes to the code, often in application-specific ways. Second, packet-based approaches break down if packets of similarly-behaving points cannot be identified, or if apparently-similar points later *diverge* as they traverse the tree (*e.g.*, because two rays reflect off a surface in different directions). In such situations, *SIMD utilization* (the fraction of useful operations performed during one SIMD instruction) drops, as most SIMD instructions end up operating on fewer than four points:

> In situations like physical simulation, collision detection or raytracing in scenes, where rays bounce into multiple directions (spherical or bumpmapped surfaces), coherent ray packets break down very quickly to single rays or do not exist at all. In the above mentioned tasks, packet oriented SIMD computations is much less useful. [36]

Clearly, it is difficult to exploit this seemingly-simple source of SIMD opportunity, even in hand-written implementations, let alone generally. As a result, most hand-written SIMD implementations focus on highly application-specific sources of vectorization. Similarly, prior attempts to generalize SIMDization of irregular applications have focused on alternative sources of vectorization opportunities [34–38].

In this dissertation, we describe *generally-applicable, systematic approaches to scheduling that expose SIMD opportunities and maximize utilization* [8]. In fact, our approaches enable packet-based SIMD computation even in algorithms where careful, hand-written implementations could not exploit it.

The problem of carefully scheduling traversals to expose opportunities for SIMDization has parallels with improving locality in tree traversal codes. The key insight is that the aforementioned locality transformations (point blocking and traversal splicing) can be readily adapted to both transform traversal codes so they can be SIMDized and schedule traversals so that SIMD utilization is maximized. Point blocking can be used to transform code to enable packetized traversals. Point blocking alone, however, is not sufficient. As described in the quote above, the divergence of points during traversals leads packet-based approaches to break down, producing very poor SIMD utilization. While our use of point blocking mitigates this effect, it does not fully ameliorate it. We show how a particular aspect of traversal splicing, its ability to dynamically sort points during traversal, can be leveraged to dramatically improve SIMD utilization. We develop an automatic transformation framework called SIMTREE[3] that leverages simple annotations to restructure a tree-traversal application so it can be readily vectorized. Our transformations can deliver significant performance gains through effective SIMDization, yielding speedups of up to 6.59 (geometric mean: 2.78).

1.4 Contributions and organization

This dissertation is the first to develop general compiler transformations and optimizations to automatically enhance the performance of tree traversal algorithms. The primary contributions are:

1. The development of *point blocking* and *traversal splicing* two new, general transformations for tree traversal codes which can effectively transform applications for enhanced locality (Chapter 3).

[3] `https://engineering.purdue.edu/plcl/simtree/`

2. The implementation of a transformation and tuning framework, TREESPLICER, that can automatically transform tree traversal algorithms to apply point blocking and traversal splicing, and autotune suitable parameters for these transformations. Experimental evidence that TREESPLICER can substantially enhance the locality and performance of tree traversal codes (Chapter 4).

3. The development of systematic transformations to both expose SIMD opportunities, through point blocking, and enhance SIMD utilization, through traversal splicing(Chapter 5).

4. The implementation of a framework, SIMTREE, for automatically restructuring traversals and data layouts to enable SIMDization. Experimental evidence that tree-traversal applications can be effectively vectorized, delivering substantial performance gains and, in some cases, exploiting more SIMD opportunities than previous manual approaches could (Chapter 6).

We start with an overview of tree traversal algorithms in Chapter 2. Then we discuss each of our contributions as described above. We discuss related work in Chapter 7, and conclude in Chapter 8.

The core contributions of this dissertation has been published, or will be published at top conferences in the area of study. Our locality work has been published at OOPSLA 2011 [6] and OOPSLA 2012 [7]. Our vectorization work will appear at PACT 2013 [8]

2. TREE TRAVERSAL ALGORITHMS

The pattern of repeated tree traversals is a recurring theme, appearing in algorithms such as Barnes-Hut [3], nearest neighbor [4], iterative closest point [39] and many ray tracing algorithms [40], among others. We adopt some unifying terminology when discussing these algorithms: *points* are the entities that traverse the tree (they may be astral bodies in Barnes-Hut, rays in ray tracing, etc.), while *nodes* are the individual elements of the tree data structures that are being traversed. Our definition of a tree traversal algorithm is thus: *an algorithm where each of a set of points recursively traverses a tree of nodes*. Note that the traversals in these algorithms are recursive, and hence depth-first.

To explain the behavior of tree traversal algorithms, we will use point correlation (PC) as an example. The two-point correlation can be calculated for a set of points by determining, for each point, p, the number of other points in the set that fall within a certain radius, r of p. PC is an important algorithm in many disciplines, such as bioinformatics and data mining [4, 38].

The naïve approach to PC would be to compare each point to every other point in the data set, an $O(n^2)$ process. To accelerate the procedure, the standard approach is to build a spatial structure over the points called a *kd-tree* [41]. This structure is built top-down: a root node is created with a bounding box that encompasses all the points. Then a *split-plane* is computed that partitions the points in the bounding box into two equal pieces, creating two children nodes for the root, each with their own bounding box. This process is repeated until the leaf nodes contain single points. Now PC can be performed by a recursive traversal of the kd-tree. Each point p starts at the root and only traverses a child if the bounding box of that child can contain points within r of p. Thus, large portions of the tree need not be traversed, reducing the overall run time. The pseudocode for this algorithm is given in Figure 2.1.

```
1 Set<Point> points = /* points */
2 Node root = buildTree(points);
3 foreach (Point p : points) {
4   recurse(p, root);
5 }

7 void recurse(Point p, Node n) {
8   if (!canCorrelate(p, n.boundingBox)) {
9     return
10  } else if (n.isLeaf()) {
11    p.updateCorrelation(n.getPoint());
12  } else {
13    recurse(p, n.leftChild);
14    recurse(p, n.rightChild);
15  }
16 }
```

Fig. 2.1. Pseudocode of point correlation

```
1 Set<Point> points = /* points */
2 foreach (Point p : points) {
3   foreach (Node n : p.oracleNodes()) {
4     visit(p, n);
5   }
6 }
```

Fig. 2.2. Point correlation as doubly nested loop

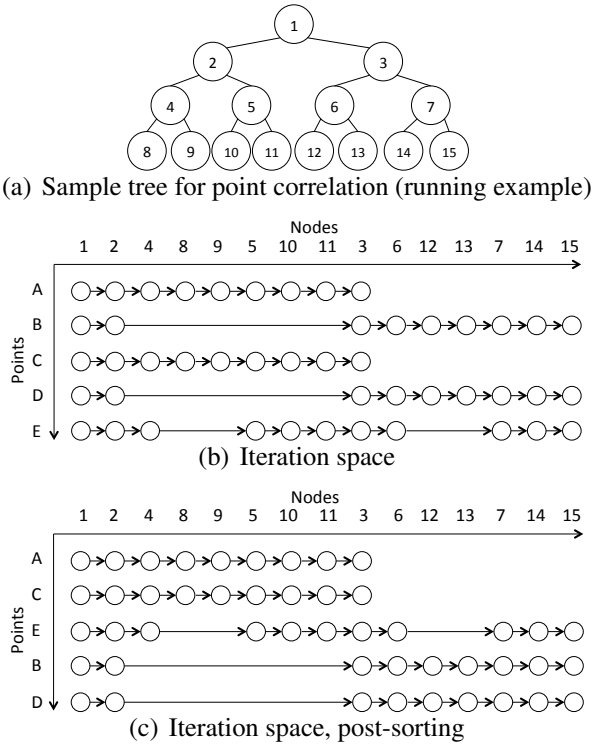
2.1 An abstract model

Reasoning about codes that traverse recursive structures is difficult for a number of reasons. First, unlike in regular applications, the structure of the key data structures is highly input-dependent. The kd-tree generated in point correlation is dependent on the particular locations of the points in the system. Furthermore, the data structures are dynamically allocated, and hence can be scattered throughout memory. Finally, the traversals are not uniform; a traversal can be truncated (*e.g.*, due to the distance check in line 8 of Figure 2.1), and traversals for two different points are not necessarily similar.

Hereon we focus on locality in tree traversals. The abstractions and iteration spaces presented here will be used to discuss vectorization in Chapter 5.

We can still reason about locality by considering the behavior of a traversal algorithm in a more abstract sense. Rather than viewing a traversal as a recursive, depth-first walk of a data structure, we can instead visualize the traversal in terms of the actual nodes touched. Fundamentally, processing a single point requires accessing some sequence of tree nodes. The particular arrangement within the tree of those nodes is irrelevant; all that matters is the ultimate sequence in which the nodes are touched. If we imagine that there is an oracle function `oracleNodes` that generates the sequence of nodes accessed while processing a particular point, we can rewrite the code of Figure 2.1 as shown in Figure 2.2 (the `visit` method abstracts whatever computations a point performs when it visits a node of the tree). In other words, we can view the algorithm as a simple, doubly-nested loop. Notably, *for the purposes of locality, the behavior of the original point correlation code is equivalent to the abstract algorithm*. All that matters is the sequence of accesses; the additional computations required to determine whether to continue a traversal or not do not affect locality. Thus, the sequences of memory accesses for the code in Figure 2.1 and Figure 2.2 are identical. Moreover, locality-enhancing transformations on the abstract code will also enhance locality in the original code, if an equivalent transformation can be applied.

Figure 2.3(a) shows a sample kd-tree for PC, with nodes numbered in heap order, and figure 2.3(b) shows an iteration-space diagram for one set of traversals; this will serve as a running example throughout the paper. Each circle in the iteration space diagram represents one dynamic instance of the loop body. The vertical axis shows the outer loop over the points, while the horizontal axis shows which nodes are visited by each point. Note that each point does not visit each node. We can use reuse distance [42] to analyze the locality behavior of the algorithm. We note that each point enjoys good locality, while each tree node has relatively poor locality: concentrating on tree node ④, we see that between point A's first access to ④ and C's reuse of ④, we must visit all 14 other nodes of the tree before we return to ④. In general, the reuse distance for most nodes will be proportional to the size of the tree.



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

(b) Iteration space

(c) Iteration space, post-sorting

Fig. 2.3. Sample tree and iteration spaces

2.2 Point sorting

While all tree traversal algorithms we consider have the same basic structure, they each traverse their trees according to different criteria and use trees with different structures (oct-trees for Barnes-Hut, kd-trees for nearest neighbor, bounding volume hierarchies for ray tracing), and dynamically allocate their trees according to input data. Hence, it appears at first glance that there may not be any unifying principles governing their locality. However, as the same tree is traversed by each point (the outer loop in Figure 2.1), there is significant data reuse. Points that are nearby in space are likely to perform very similar traversals of the oct-tree, visiting the same set of tree nodes. Thus, if these traversals are performed consecutively, the oct-tree nodes visited during the first traversal are likely to remain in cache during the second traversal, exploiting temporal locality.

Such a locality-exploiting order of traversals can be arranged by reordering the points, so that consecutive points have similar traversals. Though the optimization has only been applied to Barnes-Hut in the literature, we note that analogous transformations can be applied to any traversal code: if the points are sorted to maximize the overlap between consecutive traversals, locality can be improved.

Previous work has proposed sorting points *a priori* using application specific heuristics (*e.g.*, with a space-filling curve) [16, 17, 22]. In Section 5.3.1, we show how points can be sorted *dynamically, during traversal* without relying on application semantics. This dynamic sorting is often more effective than a priori sorting, because it can take advantage of runtime information and has more flexibility to tune the sorting.

Figure 2.3(c) shows the same traversals as Figure 2.3(b), but in a different order so that points with similar traversals are executed close together. Because points have different traversals, this sorting can reduce reuse distance; when point A and point C are executed consecutively, the reuse distance of tree node ④ drops to 8, and, in general, the reuse distance for a node will be proportional to the size of a *traversal*.

Point sorting can have substantial impact on the performance of traversal codes. Table 2.1 shows the runtimes, CPI and L2 miss rates of several benchmarks with and without sorting[1]. Sorting dramatically reduces runtime, CPI and L2 miss rates, and it is clear that we would like to employ point sorting whenever possible.

Unfortunately, point sorting has two downsides. First, *the right execution order for points is application-specific* and can require significant programmer effort to divine. Proposals that rely on point sorting adopt approaches such as using the tree-order of points [17] or space-filling curves [16, 22]. Choosing the appropriate order for a given algorithm requires deep knowledge of the algorithm's behavior; this is especially problematic for algorithms such as nearest neighbor, where different points traverse the tree in different orders. In Section 3.2, we introduce a new locality optimization that performs this sorting "on-the-fly" and does not require any application-specific knowledge to implement.

[1]Results are obtained via PAPI, with the random inputs on the Opteron II machine, described in more detail in Section 4.6.1.

Table 2.1
Efficacy of sorting

Benchmark	Runtime (sec)		CPI		L2 miss rate	
	Unsort	Sort	Unsort	Sort	Unsort	Sort
Barnes-Hut (BH)	123.6	35.3	4.47	1.25	0.501	0.005
Point Correlation (PC)	895.2	327.1	5.21	1.50	0.486	0.571
Nearest Neighbor (NN)	380.1	175.7	3.02	1.35	0.667	0.445
k-Nearest Neighbor (kNN)	818.2	273.9	6.13	2.17	0.688	0.614
Ball Tree (BT)	1102.0	423.3	3.83	1.66	0.651	0.571
Ray Tracing (RT)	166.4	104.0	2.71	1.64	0.450	0.192

Table 2.2
Limitations of sorting for large traversals

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

Second, the sorting optimization loses its effectiveness as the traversal sizes get larger. With a sufficiently large traversal, the least recently visited nodes of the oct-tree will be evicted from cache, and hence when the next point is processed those nodes will have to be brought back in to cache, incurring additional misses. Table 2.2 shows, for several tree sizes, the average traversal size, the L2 miss rate of an optimized implementation, and the % improvement in cycles over an un-optimized implementation. The test system is a dual-core Intel Pentium with 32K L1 data cache per core and 1M shared L2 cache. The efficacy of sorting is clear for small sizes: in an input with 10,000 points, the sorting optimization improves runtime by 67%. However, with an input of 1 million points, the sorting optimization has much higher miss rates, and only improves runtime by 26% compared to the un-optimized version. Clearly, a more sophisticated optimization is necessary to continue exploiting locality as traversal sizes get larger.

3. ENHANCING LOCALITY FOR TREE TRAVERSAL ALGORITHMS

3.1 Point blocking

Though sorting is a useful optimization that reduces the reuse distance for tree nodes to be on the order of traversal size, it loses its effectiveness when inputs, and hence traversal sizes, get too large. In this section we introduce *point blocking* as a method for improving locality for tree traversal codes when traversal sizes attenuate the benefits of sorting [6].

Given the abstract, outer-product model of traversal codes described in Figure 2.2, several analogs of classical loop transformation techniques become apparent. For example, loop interchange would place the traversal loop on the outside, with the point loop on the inside. This corresponds to choosing a node of the recursive data structure, then processing each point that must interact with it. However, we note that, just as the original code suffers from poor locality if the traversal vector exceeds cache, the loop-interchanged code will suffer from poor locality if the point vector exceeds cache. For large inputs, this is likely[1].

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

```
1 Set<Block<Point>> blocks = /* points */
2 foreach (Block<Point> b : blocks) {
3   foreach (Node n : b.oracleNodes()) {
4     foreach (Point p : b.validPointsAt(n) {
5       visit(p, n);
6     }
7   }
8 }
```

Fig. 3.1. Point blocked abstract code

While loop interchange may not produce an effective implementation of a traversal code, loop tiling holds promise. In particular, we propose tiling the point vector, which produces the code seen in Figure 3.1. Essentially, this code breaks the points into blocks of size B. For each block, each node of the recursive data structure is chosen, then each point in the block is processed for the chosen node. If B is chosen correctly, the points in a block will never leave cache. Further, regardless of how large the traversal is, each node of the traversal will only incur a cache miss once per block[2]. We call this transformation *point blocking*.

3.1.1 Concrete implementation of point blocking

The concrete pseudocode to realize this tiling is shown in Figure 3.2. Compared to the base algorithm of Figure 2.1, the code is similar except the recursive method operates over a block of points rather than a single point. Rather than finishing an entire traversal for one point before moving on to the next point, a block of points moves through the tree in lockstep. The block visits nodes in the tree comprising the union of its component points' traversals, and points within the block only interact with nodes they would have in the original code. If none of the points in a block interact with a node, the block will skip visiting the node. The arrows in Figure 3.3 show the new iteration order when applying point blocking to the sorted traversals of Figure 2.3(c), with block size 3. Note that the tree nodes enjoy improved locality: they will incur misses once per block, instead of once per point. Further, the reuse distance of a point is on the order of the block size, so as long as blocks are properly sized, points will suffer only cold misses.

3.1.2 Limitations of point blocking

We note, however, that point blocking's effectiveness relies on point sorting as a pre-processing pass. A more precise characterization of the locality behavior of a point blocked

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

```
1 Set<Point> points = /* points */
2 Node root = buildTree(points);
3 foreach (Block<Point> b : points) {
4   recurse(b, root);
5 }

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

Fig. 3.2. Concrete code for point blocking

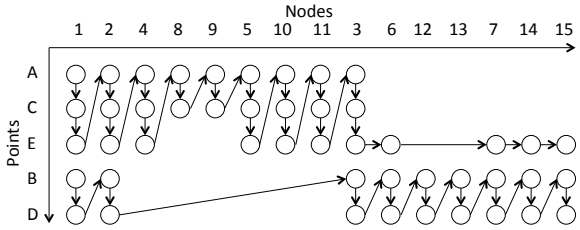


Fig. 3.3. Iteration order for point blocking

code is that a tree node suffers one miss *per block that visits it*. If the points are sorted, then points that visit a particular node are likely to be collected into a relatively small number of blocks, and hence the node will suffer few misses. If the points are unsorted, each block will have to visit more tree nodes (as its points' traversals will have less overlap and hence

Table 3.1
Dependence of point blocking on point sorting

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

cover more ground). Thus, each tree node is visited by more blocks, and will suffer more misses. Consider node ④ from our running example. In the sorted version of the point-blocked code (Figure 3.3), all the points that visit ④ are in the same block, and ④ only suffers a single miss. However, if we were to apply point blocking to the original order of points from Figure 2.3(b), we note that two blocks would have to visit ④, resulting in two misses.

Table 3.1 shows the runtimes, CPI and L2 miss rates of several benchmarks with point blocking, with and without sorting[3]. While point blocking can often improve performance for both scenarios with and without sorting, the gap between sorted point blocking and unsorted point blocking remains significant. Hence we would like to combine point blocking with point sorting to achieve the best performance. Unfortunately, as discussed before, performing sorting requires application-specific knowledge, and a general transformation cannot rely on having sorted inputs. In the next section, we introduce a transformation whose performance is less dependent on the order in which points are processed, obviating the need for application-specific sorting.

[3]Also with the random inputs on the Opteron II machine, as in Table 2.1.

3.2 Traversal Splicing

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

The most intuitive way to visualize traversal splicing is that, rather than tiling the point loop, as in point-blocking, it tiles the traversal loop, yielding the abstract pseudocode of Figure 3.4(a). Thus, rather than picking a block of points and following their traversals through the entire tree, traversal splicing takes a single point and executes a *partial traversal* of the tree. It then takes the next point and executes a partial traversal and so on. Once each point has executed a partial traversal, the first point's traversal picks up from where it left off. This process can be extended by dividing each traversal up into several partial traversals, whose executions are interleaved. In essence, each point's traversal is chopped up into pieces, and the pieces are rearranged and stitched together in a different order; hence, *traversal splicing*. The nodes at which the traversals are paused are called *splice nodes*.

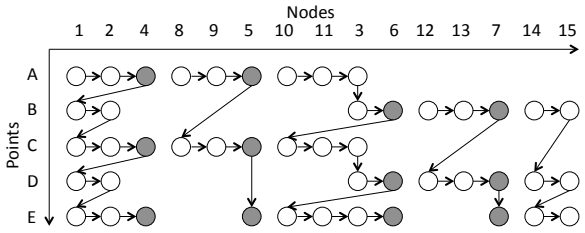
Figure 3.4(b) shows the effects that traversal splicing has on the iteration space from Figure 2.3(b), with the iterations that visit the splice nodes (④, ⑤, ⑥ and ⑦) filled in[4]. Note that the partial traversals are executed in "lock-step," and if a point does not encounter a splice node (consider point B, which does not visit node ④ or ⑤), it resumes once all other points have arrived at the next node it should visit.

We can use the iteration space diagram to reason about the locality effects of traversal splicing. We note that each node in the tree has good locality. The reuse distance of a tree node is bounded by the distance between splice nodes. As long as the splice nodes are not too far apart (*i.e.,* each set of nodes in line 3 of Figure 3.4(a) fits in cache), we get only

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

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

(a) Abstract pseudocode for traversal splicing



(b) Iteration order for traversal splicing

Fig. 3.4. Traversal splicing

cold misses on the tree nodes. The points, however, will miss once per partial traversal (when a point pauses at a splice node, it will not be re-accessed until every other point has completed a partial traversal). Interestingly, the order of points is irrelevant to this locality analysis. Hence, *traversal splicing is agnostic to whether or not the points are sorted.*

3.2.1 Dynamic sorting

Note that the above analysis assumes that splice nodes are not too far apart. Unfortunately, the structure and size of the tree can be extremely irregular, leading to sets of nodes (line 3 of Figure 3.4(a)) that may exceed cache. However, the *partial traversals* of points (line 5 of Figure 3.4(a)) may still fit in cache. In this situation, the order of points once again matters: points with similar partial traversals between splice nodes can enjoy good locality if they are processed consecutively. But how can this reordering be done automatically?

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

As an example, consider applying this strategy to the traversals of Figure 3.4(b). We will process the points in their original order until splice node ④. Because points B and D did not reach node ④, they will be reordered with respect to points A, C and E. The order in which the points will be processed for the partial traversals between ④ and ⑤ is (A, C, E, B, D). Note that this new order is precisely the order in which the points would have been executed had they been sorted *a priori* (see Figure 2.3(c)).

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

3.2.2 Combining traversal splicing and point blocking

Finally, if we consider the inner two loops of Figure 3.4(a), we note that it looks like the abstract version of the original traversal code. If the partial traversals become too large, it is clear that applying point blocking to the inner two loops (which, recall, are now operating over dynamically sorted points) can further enhance locality. Section 4.5 discusses how applying point blocking to partial traversals allows our transformed code to tolerate larger partial traversals, and hence reduces its sensitivity to splice node placement. Section 6.2

quantifies the performance of point blocking and traversal splicing individually, and the combination of the two.

3.2.3 More complex traversals

Matters are more complicated when traversals of points do not take the same path through the tree. In PC, each point traverses the tree in the same order, aside from truncations; there is a single global traversal order, and each point's traversal is a filtered subset of that order. Figure 3.5 shows the pseudocode for nearest neighbor (NN), in which the traversal order is not fixed. For example, at a given node, some points may visit the node's left child before its right, while other points visit its right child before its left.

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

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

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

```
1 Set<Point> points = /* points */
2 Node root = buildTree(points);
3 foreach (Point p : points) {
4   recurse(p, root);
5 }

7 void recurse(Point p, Node n) {
8   if (!canBeCloser(p, n.boundingBox)) {
9     return;
10  } else if (n.isLeaf()) {
11    p.updateClosest(n.getPoint());
12  } else {
13    double split = p.value(n.splitType);
14    if (split <= n.splitValue) {
15      recurse(p, n.leftChild);
16      recurse(p, n.rightChild);
17    } else {
18      recurse(p, n.rightChild);
19      recurse(p, n.leftChild);
20    }
21  }
22 }
```

Fig. 3.5. Pseudocode of nearest neighbor

of the splicing procedure for applications like PC. In PC, because each point follows the same path through the tree, every point's ith phase ends at the same splice node.

The only complication is when a point is truncated before reaching a splice node, skipping one or more splice nodes. In this case, the truncated point conceptually still participates in that splice node's phase, but without performing any work; the point's traversal will resume when the phases for any skipped splice nodes are over. Section 4.4 describes this phasing algorithm, and how it can be scheduled efficiently, in more detail.

3.2.4 Splice node placement

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

1. If different points exhibit different traversal orders, the phasing algorithm outlined above requires that each point that will encounter a splice node in phase i encounter that phase's splice node at the same time. This can easily be accomplished by placing all splice nodes at a uniform depth from the root node . Section 3.2.5 discusses how this criterion can be relaxed.

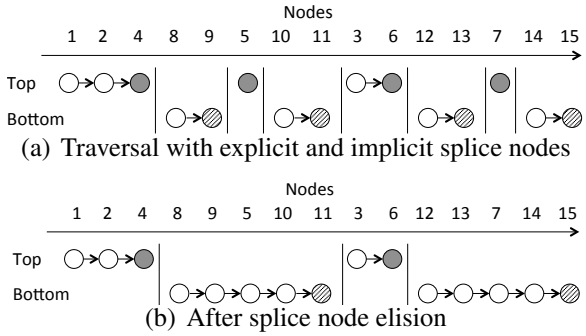
2. The deeper in the tree splice nodes are placed, the more splice nodes, and hence partial traversals, there are. Placing splice nodes too deep can result in high overhead (due to the extra bookkeeping necessary to keep track of partial traversals) and poor locality (as points incur a miss once per partial traversal).

3. Conversely, if the points are too high in the tree, then too many points will reach the same splice nodes, reducing the efficacy of the reordering optimization. Further, the portions of the traversal "below" the splice nodes will be large, potentially resulting in poor locality, though this effect can be mitigated by applying point blocking, as discussed in Section 3.2.2

Good splice node placement requires striking a balance between placing the nodes too shallow in the tree for reordering to be useful and placing them too deep to take advantage of reordering. Section 4.5 describes our approach to splice node selection.

3.2.5 Optimizations

Traversal splicing as described in Section 3.2 comes at a cost. It requires that traversals be paused and resumed at splice nodes. In principle this would require maintaining the full stack for each point's traversal, allowing it to be paused at a splice node and its continuation resumed in the next phase. Rather than storing a point's stack in some ancillary data



(a) Traversal with explicit and implicit splice nodes

(b) After splice node elision

Fig. 3.6. Top and bottom phases

structure, we record the information in the tree itself. When a point is at some node n in its traversal, each level of its stack can be stored in the appropriate ancestor of n. Hence, at each node in the tree, we will store, per point, any information needed by the point at that level in its recursion. This includes any local variables of the recursive method, as well as a program counter, recording where in the recursive method the point was when it descended from the node. This information needs to be tracked for each point, and can consume space proportional to the depth of any traversal. Because all points are in flight simultaneously, the amount of extra space required per point can lead to prohibitive overheads for traversal splicing.

This section discusses optimizations that take advantage of *structural characteristics* of the target tree traversal algorithm that allow traversal splicing to be implemented with far less space overhead. In the following presentation, D refers to the depth of the splice nodes (recall from Section 3.2.4 that we place all splice nodes at a uniform depth from the root).

Implicit splice nodes

To avoid storing stack information at *every* node along a point's traversal, we note that partial traversals that start after a splice node (*e.g.*, the partial traversals starting after node ⑤ in Figure 3.4(b)) visit the entire subtree of the splice node before returning higher in the tree. We introduce the concept of *implicit* splice nodes: nodes that act as splice nodes for

the purposes of pausing and restarting traversals, but are not explicitly marked by the transformation. The last node visited by any partial traversal in the subtree "below" an explicit splice node is an implicit splice node. Figure 3.6(a) shows the resulting decomposition of a traversal over the tree in Figure 2.3(a). The explicit splice nodes are shaded in gray, while the implicit splice nodes are shaded with diagonal lines. We can categorize the partial traversals as "top" traversals that traverse the top portions of the tree and end at an explicit splice node, and "bottom" traversals that traverse the lower portions of the tree and end at an implicit splice node. Notably, the bottom traversals are equivalent to normal recursive traversals over the subtrees rooted at the explicit splice nodes. Therefore, we need only track information for traversal splicing for top phases. Because any top phase is no larger than a single path from the root to a splice node, the maximum amount of space required to track each point is now $O(D)$.

Reducing stack storage

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

To track a point's program counter efficiently, we group each straight-line series of calls in a pseudo-tail recursive method into a *call set*. Each call set has a fixed sequence of recursive calls (*i.e.*, a fixed order of visiting a node's children), so a point's behavior at this

node is completely determined by which call set it uses, and the call set it uses is computed before the first recursive call.

The nearest neighbor (NN) code of Figure 3.5 shows an example of a pseudo-tail recursive method with multiple call sets. Here the two possible call sets (traversal orders) are {leftChild, rightChild} (lines 15 & 16), and {rightChild, leftChild} (lines 18 & 19), and which call set a point will take is decided before any recursive calls are made. Thus, at each level, a point need only track which call set it used, and where in the call set it was, to fully reconstruct its stack. The phased nature of the traversal splicing algorithm means that every point's location in their respective call sets will be aligned; at any given time, every point will be executing the first recursive call of their call set, or every point will be executing the second recursive call of their call set, etc. Hence, at each depth we can track a global "phase number" that maintains where in its call set each point is. A point only needs to maintain which call set it is using at a given level.

Inferred order

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

Note further that because the data structure being traversed is a tree, knowing where a point is in the tree at any given time during execution uniquely determines the path from the root to that point. Hence at a given level we need not store which child the point visited, either. This eliminates the need to track any information per point aside from a global phase number per level (shared across all points) and the current tree node the point is at. This reduces storage needs to $O(1)$ per point. A special case of inferred order is when there is a single call set in a recursive method as in PC (Figure 2.1(a)).

Splice node elision

Given our policy of placing splice nodes at a fixed depth, certain top phases, which begin just above the splice depth, will necessarily be very short. For example, the partial traversal starting at node ⑤ is only one node long! To avoid the overhead of performing splicing when it is unlikely to be effective, we *elide* splice nodes for short phases. That is, if a partial traversal is likely to be short, we combine it with the following partial traversal. In practice, for top phases that begin a short distance above the splice depth D, we do not perform splicing and instead immediately begin a bottom phase, as shown in Figure 3.6(b). This can be both good and bad for locality. It is good as we incur fewer misses on each point (as we have fewer phases), but the combined bottom phase becomes larger, potentially outstepping cache. When splice node elision is combined with point blocking, the latter effect is mitigated. Section 4.5.2 discusses our strategy for splice node elision.

4. AUTOMATICALLY ENHANCING LOCALITY WITH TREESPLICER

In the previous sections we have discussed how loop transformations can significantly reduce cache misses in codes that repeatedly traverse trees. Realizing these transformations is non-trivial because each point may have a different traversal (*i.e.*, each point may require traversing a different portion of the data structure). We must ensure that these differing behaviors are respected by the transformation. In this section, we describe an analysis and transformation framework, called TREESPLICER, which can apply the transformations automatically. TREESPLICER is written as a series of passes in the JastAdd framework [43], which enables analysis and transformation of Java programs. We have made the source code of TREESPLICER public at `https://engineering.purdue.edu/plcl/treesplicer/`.

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

1. *Identifying repeated traversals.* TREESPLICER finds possible transformation opportunities by looking for code that performs repeated traversals of recursive structures. (Section 4.1).

2. *Verifying correctness.* TREESPLICER analyzes the dependences in the identified loop to determine whether the transformations can be legally applied. (Section 4.2).

3. *Applying point blocking.* If the transformations are legal, TREESPLICER automatically applies point blocking (Section 4.3) and/or traversal splicing (Section 4.4).

4.1 Identifying opportunities for transformation

While recognizing a traversal code structure can be expedited with programmer annotations, many traversal codes have a common algorithmic structure that does not require annotations to recognize. In particular, many traversal codes are written by recursive function calls on recursive data structures. If an application performs *repeated recursive traversals of a recursive structure*, TREESPLICER will identify it as a candidate for point blocking.

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

This doubly recursive structure is illustrated for PC in Figure 2.1. The node class `Node` is a recursive class with fields `leftChild, rightChild` that are also of type `Node`. The method `recurse` (lines 7-16) is a recursive method that takes a recursive class as an argument. The traversal is realized by calling `recurse` on the children of a node (lines 13-14). A termination condition stops the recursion if the point is far enough away from the node (line 8).

The algorithmic structure that we want to identify must be a combination of both *recursive method calls* and *recursive structures*. Recognizing each individually is trivial. A recursive method, m, can be recognized by finding a call to itself within a method's body[1]. A recursive structure can be recognized by finding a class, c, with at least one field f of the same class (or superclass).

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

```
1 Set<Point> points = /* points */
2 Node root = buildTree(points);
3 foreach (Point p : points) {
4   root.recurse(p);
5 }

7 class Node {
8   Node leftChild, rightChild;
9   void recurse(Point p) {
10    if (!canCorrelate(p, boundingBox)) {
11      return;
12    } else if (isLeaf()) {
13      p.updateCorrelation(getPoint());
14    } else {
15      leftChild.recurse(p);
16      rightChild.recurse(p);
17    }
18  }
19 }
```

Fig. 4.1. Passing recursive class via implicit argument

We must then determine whether the recursive method performs a recursive traversal of any identified recursive structures. This might happen in one of two ways: (i) if m takes an object o of class c as an argument, and passes $o.f$ as an argument to the recursive call; or (ii) if m is a member method of c and it performs the recursive call by invoking $f.m()$ (in other words, the data structure node is the implicit "this" argument). The former was illustrated in Figure 2.1. Figure 2.1 could be re-written to Figure 4.1, where the explicit argument child in lines 13-14 of Figure 2.1 has changed to the implicit argument in lines 15-16 of Figure 4.1. Using implicit arguments is common programming style, and TREESPLICER handles both cases.

Having identified a recursive traversal of a recursive structure, TREESPLICER's next goal is to determine if it is repeated. To do so, TREESPLICER uses a call graph analysis to determine that the recursive method is called (either directly, or through a chain of calls) from a loop in the application. Our transformations are useful when the traversals are

repeated many times. If the number of loop iterations can be statically determined to be larger than N, TREESPLICER transforms the code. If the number of loop iterations is decided at runtime, TREESPLICER inserts run-time profiling code that uses the transformed path if there N or more iterations. We empirically choose $N = 10000$. TREESPLICER will perform the transformations for every kernel it identifies that is a *repeated recursive traversal of a recursive data structure*, which satisfies the correctness conditions described in the next section.

4.2 Correctness

As in any loop transformation in regular programs, our transformations must preserve dependences to ensure correctness.

4.2.1 Correctness for point blocking

We will refer to the iteration space of point blocking, which was depicted in Figure 3.3. While the transformed code walks the iteration space in a different order than the original code, a few key aspects of the execution order are preserved. First, for a given point, nodes are visited in the same order. Hence, intra-point dependences (dependences that point "right" in the iteration space) are preserved. Second, if the data structure being traversed is a tree, for a given node, points "visit" the node in the same order. Hence, while not all inter-point dependences are respected by the transformation, intra-node dependences, where values on a node are updated as it is visited by points, are preserved as well.

Other types of dependences are not preserved. For example, inter-point *and* inter-node dependencies (*e.g.* processing point C at node ④ depends on processing point A at node ⑧) which were legal in the original code may not be preserved in the transformed code. Further, if i) the structure being traversed is not a tree or ii) there are multiple orders in accessing children (as discussed in Section 4.3.2), there may be multiple paths to reach a certain node, or a traversal may access a node multiple times; in these situations, inter-point

dependences (even those that are intra-node) may not be preserved by the transformation. Section 4.2.3 discusses some of the implications of traversals of non-tree data structures.

Point blocking relies on programmer annotations which specify that the point loop is parallelizable, as a conservative guarantee that there are no problematic inter-point dependencies[2]. Parallelizability is a *sufficient* condition for point blocking to be legal, not a necessary one; tree traversals where values on nodes are updated as points visit them can be transformed, but not parallelized.

4.2.2 Correctness for traversal splicing

Traversal splicing is the equivalent of "tiling" the traversal loop in the doubly-nested loop formulation of tree traversal algorithms shown in Figure 2.1(b), and the correctness criteria for loop tiling (that the loop be fully permutable) still apply. In particular, the order in which a point visits nodes in its traversal is unchanged, as is the order in which a given tree node is visited by different points. Thus, traversal splicing can be performed in the presence of intra-traversal dependences (*e.g.*, if some data associated with the point is updated at each leaf node of the point's traversal) or dependences that cross traversals but stay within the same node (*e.g.*, if a counter at each node is updated whenever a point visits the node).

When reordering is performed during splicing, the correctness criteria change. Each point's traversal still occurs in the prescribed order, and hence intra-traversal dependences are preserved. However, because the order in which partial traversals happen can be shuffled around, inter-traversal dependences are no longer guaranteed to be preserved. Point blocking's sufficient condition of a parallelizable point loop is also sufficient to ensure correctness of traversal splicing, with or without dynamic sorting.

[2]Proving that there the point loop is parallelizable likely requires a shape analysis and is beyond the scope of this work.

4.2.3 Discussion: Generalization to DAGs and graphs

TREESPLICER identifies data structure traversals by looking for recursive traversals of recursive structures. While the discussion so far has focused on TREESPLICER's application to traversals of trees, the framework's identification strategy may actually flag traversals of DAGs and general graphs as potential optimization targets, as well. Interestingly, the transformations presented here can be applied directly to these more general data structures. From the perspective of traversal algorithms, the key distinction between trees and the more general structures is that, in the latter case, depth-first traversals may visit the same node more than once. This means the correctness criteria are more complex than for trees. Nevertheless, TREESPLICER's sufficient condition of looking for parallelizable loops means that there will be no inter-traversal dependences, so point blocking will be correctly applied.

However the recursion unrolling of traversal splicing requires that the data structure be a tree, and is not applicable to more general structures such as DAGs or graphs. Hence traversal splicing relies on annotations which specify that the data structure is a tree, in addition to point loop parallelizability for correctness. We leave the problem of checking more complex correctness conditions, as well as an investigation of the efficacy of point blocking for DAG and graph traversals to future work.

4.3 Implementation of point blocking

Once we have identified the recursive structures that can be transformed correctly, the next step is to realize the transformation efficiently. This section discusses the actual implementation of point blocking. We will first explain how TREESPLICER operates in the simple case: code such as PC (Point Correlation). We will then describe how TREESPLICER handles complications in the basic algorithmic pattern: multiple recursive calls within a method, different arguments to recursive calls, and chains of method calls between the loop and the recursive method.

```
1 Point [] points = /* entities in algorithm */
2 Object o1 = /* something loop invariant */
3 for(int i = 0; i < points.length; i++) {
4   Point p = points[i];
5   Object o2 = /* something loop variant */
6   // do something - prologue
7   recurse(p, o1, o2, root);
8   // do something - epilogue
9 }

11 void recurse(Point p, Object o1, Object o2, Node node) {
12   // do something
13   if (cond) {
14     foreach (Node child : node.children) {
15       recurse(p, o1, o2, child);
16     }
17   }
18 }
```

Fig. 4.2. Generic structure - original code

4.3.1 Basic transformation

We start with a simple, generic, repeated recursive traversal of a recursive structure as in Figure 4.2. Our analysis will find a recursive method (lines 11-18) associated (explicitly or implicitly) with a recursive class, and an enclosing loop (lines 3-9). There can be arbitrary code within the enclosing loop, and arbitrary arguments can be passed to the recursive method. The generic code shows two such arguments, one that is loop variant and another that is loop invariant with regard to the enclosing loop. This distinction is necessary because loop variants need to be saved in the block. The type of the arguments is irrelevant, they are deemed Object for the sake of illustration. The transformed code of the generic recursive structure is shown in Figure 4.3. The automatically generated block classes corresponding to the transformed code are shown in Figure 4.4. We will now discuss the transformation step by step.

```
1 Point [] points = /* entities in algorithm */
2 Object o1 = /* something loop invariant */
3 Block b = /* block instance */
4 BlockStack stack = /* stack instance */
5 // autotuning code to be added here
6 for (int ii = 0; ii < points.length; ii+=B) {
7   for (int i = ii; i < ii + B; i++) {
8     Point p = points[i];
9     Object o2 = /* something loop variant */
10    // do something - prologue
11    b.add(p, o2);
12  }
13  stack.set[0].block = b;
14  recurse(o1, root, stack, 0);
15  for (int i = 0; i < B; i++) {
16    Point p = b.p[i];
17    Object o2 = b.o2[i];
18    // do something - epilogue
19  }
20  b.recycle();
21 }

23 void recurse(Object o1, Node node, BlockStack stack, int level) {
24  BlockSet bset = stack.set[level];
25  Block b = bset.block;
26  Block nextB = bset.nextBlock;
27  nextB.recycle();
28  for (int i = 0; i < b.size; i++) {
29    Point p = b.p[i];
30    Object o2 = b.o2[i];
31    // do something
32    if (cond) {
33      nextB.add(p, o2);
34    }
35  }
36  if (nextB.size > 0) {
37    stack.set[level + 1].block = nextB;
38    foreach (Node child : node.children) {
39      recurse(o1, child, stack, level + 1);
40    }
41  }
42 }
```

Fig. 4.3. Generic structure - transformed code

```
1  class Block {
2    int size;
3    Point [] p;
4    Object [] o2;

6    void add(Point p_, Object o2_) {
7      p[size] = p_;
8      o2[size] = o2_;
9      size++;
10   }

12   void recycle() {
13     size = 0;
14   }
15 }

17 class BlockSet {
18   Block block;  // just reference
19   Block nextBlock = /* actual allocation */
20   // more nextBlocks if necessary
21 }

23 class BlockStack {
24   BlockSet [] set;
25 }
```

Fig. 4.4. Implementation of block classes

The first step in tiling a recursive code is to transform the enclosing loop to be over blocks of points, rather than single points. In the original code, each point is processed by calling `recurse` on the root node. In the transformed code, points will be added to a block instead (line 11); once this block is full, a modified version of `recurse` will be called *on the entire block* (line 14). Note that this block may contain more than just the point; any loop-variant arguments to `recurse` (such as `o2`) are also placed in the block. Loop invariant arguments (such as `o1`) are passed to `recurse` without change. The block `b` is recycled after it has been processed (line 20). Recycling a block, simply sets its size to 0, so that the next invocation of *add* will overwrite previous points.

The enclosing loop can have arbitrary code before (prologue, line 6) and after (epilogue, line 8) the method call. This code must be split into prologues which execute for all points in the block before `recurse` (lines 7-12) and epilogues which execute for all points in the block after `recurse` (lines 15-19). Prologues and epilogues generalize to intermediary methods, discussed further in Section 4.3.4.

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

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

Figure 4.4 shows the implementation of the automatically generated block classes corresponding to Figure 4.2. The Block class allocates space for the loop variant arguments passed to the recursive method, as well as the return values of all intermediary methods. The BlockSet class has a reference to the current block, and allocates space for the next blocks of the next level. The BlockStack class is simply an array of BlockSets.

4.3.2 Handling multiple recursive calls

Interestingly we may need multiple blocks for the next level. This is because in some algorithms, points access children in different orders during their depth-first traversal. For example, this situation arises in the Nearest Neighbor benchmark, and is illustrated in Figure 4.5. Figure 4.5(a) shows the original recursive method, and Figure 4.5(b) shows how it should be transformed. The order in which the children are processed can be either left first then right, or right first then left, depending on the point. The transformed code must honor both orders, by having two next blocks. Points that take the first traversal order are added to the first block, while those that take the second traversal order are added to the second block. At the end of the current level, both next blocks are processed in the appropriate order.

It may seem that we should have a next block for each recursive invocation in the transformed method. For example, in the generic code of Figure 4.2, where there are an arbitrary number of children, it seems that we might need an arbitrary number of blocks. The crucial difference between the scenario in Figure 4.2 and the scenario of Figure 4.5 is that the latter has a divergence of control flow dependent on the point's properties. Hence, the traversal orders may differ on a point-by-point basis, and separate blocks are necessary to differentiate between the orders. In the generic scenario, lines 14–16 in Figure 4.2 will be executed for *all* points in the block, hence it can be replaced with a single block *add* call (line 33 of Figure 4.3).

TREESPLICER decides the number of next blocks required by starting at each recursive method call site within the recursive method body, and expanding the call site to the largest control flow block that is control independent of the point. Each expanded call site requires its own next block. The required number of next blocks is synthesized in the block set class (Figure 4.4), and each expanded call site is replaced with a call to add points to the associated next block. At the end of the recursive method, each next block is recursed upon if not empty.

```
1  void recurse(Point p, Node n) {
2    boolean cond = /* something dependent on p */
3    if (cond) {
4      recurse(p, n.left);
5      recurse(p, n.right);
6    } else {
7      recurse(p, n.right);
8      recurse(p, n.left);
9    }
10 }
```

(a) Original code

```
1  void recurse(Node n, BlockStack stack, int level) {
2    BlockSet bset = stack.set[level];
3    Block b = bset.block;
4    Block nextB = bset.nextBlock;
5    Block nextB2 = bset.nextBlock2;
6    nextB.recycle();
7    nextB2.recycle();
8    for (int i = 0; i < b.size; i++) {
9      Point p = b.p[i];
10     boolean cond = /* something dependent on p */
11     if (cond) {
12       nextB.add(p);
13     } else {
14       nextB2.add(p);
15     }
16   }
17   if (nextB.size > 0) {
18     stack[level + 1].block = nextB;
19     recurse(p, n.left, level + 1);
20     recurse(p, n.right, level + 1);
21   }
22   if (nextB2.size > 0) {
23     stack[level + 1].block = nextB2;
24     recurse(p, n.right, level + 1);
25     recurse(p, n.left, level + 1);
26   }
27 }
```

(b) Transformed code

Fig. 4.5. Traversing different orders of children

4.3.3 Handling different arguments

So far the transformation has assumed that recursive calls have identical arguments other than the node argument. In general, recursive calls can have different arguments as in Figure 4.6. The first recursive call takes `o1` as the last argument (line 6), and the second recursive call takes `o2` (line 7), and both `o1` and `o2` are dependent on the point. In this case, both `o1` and `o2` must be saved into the next block (line 15), and an additional `callIndex` argument is passed to decide which of `o1` and `o2` to use (lines 9–10).

4.3.4 Handling intermediary methods

In general, there can be an arbitrary number of intermediary methods from the enclosing loop to the recursive method. Handling intermediary methods in point blocking requires splitting each such method into prologues and epilogues, and saving local variables defined in the prologue and used in the epilogue.

The original and transformed code for a recursive structure with one intermediary method *foo* is shown in Figures 4.7- 4.8. The `recurse` method is the same as in Figure 4.2. The intermediary method `foo` has two local variables for illustration, `o3` which is defined in the prologue and used in the epilogue, and `o4` which is *not* used in the epilogue. Because the epilogue is executed after the prologue *for all points in the block*, local variables declared in the prologue and used in the epilogue must be saved in additional space proportional to block size. This is realized by an `IntermediaryState` class which is passed as an argument to the intermediary methods. Saving intermediary state is done at the *end* of each prologue, and loading intermediary state is done at the *start* of each epilogue. While Figure 4.7 shows a single intermediary method, the transformation generalizes to an arbitrary number of such methods.

The *last* intermediary method, which calls `recurse`, saves the points and any loop variant arguments to the block. Loop variant arguments at the enclosing for loop propagate to intermediary methods. For example in `foo`, a local variable `pp` is dependent on `p` which

```
1 void recurse(Point p, Node n, Object o) {
2   Object o1 = /* something dependent on p */
3   Object o2 = /* something dependent on p */
4   boolean cond = /* something dependent on p */
5   if (cond) {
6     recurse(p, n.left, o1);
7     recurse(p, n.right, o2);
8   }
9 }
```

(a) Original code

```
1 void recurse(Node n, BlockStack stack, int level, int callIndex) {
2   BlockSet bset = stack.set[level];
3   Block b = bset.block;
4   Block nextB = bset.nextBlock;
5   nextB.recycle();
6   for (int i = 0; i < b.size; i++) {
7     Point p = b.p[i];
8     Object o;
9     if (callIndex == 0) o = b.o1[i];
10    else o = b.o2[i];
11    Object o1 = /* something dependent on p */
12    Object o2 = /* something dependent on p */
13    boolean cond = /* something dependent on p */
14    if (cond) {
15      nextB.add(p, o1, o2);
16    }
17  }
18  if (nextB.size > 0) {
19    stack[level + 1].block = nextB;
20    recurse(p, n.left, level + 1, 0);
21    recurse(p, n.right, level + 1, 1);
22  }
23 }
```

(b) Transformed code

Fig. 4.6. Handling different arguments

```
1 Point [] points = /* entities in algorithm */
2 Object o1 = /* something loop invariant */
3 for(int i = 0; i < points.length; i++) {
4   Point p = points[i];
5   Object o2 = /* something loop variant */
6   // do something - prologue
7   Object ret = foo(p, o1, o2);
8   // do something - epilogue
9 }

11 Object foo(Point p, Object o1, Object o2) {
12   Object o3 = /* defined in prologue and used in epilogue */
13   Object o4 = /* defined in prologue but not used in epilogue */
14   Object pp = /* something dependent on p */
15   Object oo1 = /* something dependent on o1 */
16   Node root = /* something loop invariant */
17   // do something - prologue
18   recurse(pp, oo1, o2, root);
19   // do something - epilogue
20   return ret;
21 }
```

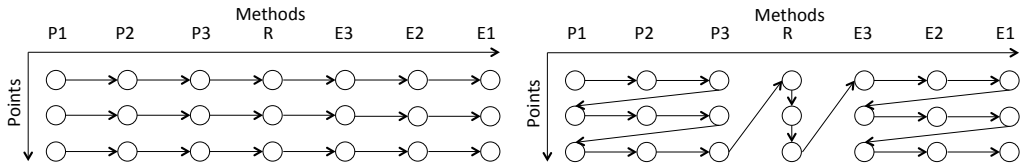
Fig. 4.7. Intermediary methods - original code

```
1 Point [] points = /* entities in algorithm */
2 Object o1 = /* something loop invariant */
3 Block b = /* block instance */
4 BlockStack stack = /* stack instance */
5 IntermediaryState interState = /* intermediary state instance */
6 interState.block = b;
7 // autotuning code to be added here
8 foo_map(o1);
9 for (int ii = 0; ii < points.length; ii+=B) {
10   for (int i = ii, interState.index = 0; i < ii + B; i++, interState.index++) {
11     Point p = points[i];
12     Object o2 = /* something loop variant */
13     // do something - prologue
14     foo_prologue(p, o1, o2, root, interState);
15   }
16   stack.set[0].block = b;
17   recurse(Block.oo1, Block.root, b);
18   for (int i = 0, interState.index = 0; i < B; i++, interState.index++) {
19     Point p = b.p[i];
20     Object o2 = b.o2[i];
21     Object ret = foo_epilogue(p, o1, o2, root, interState);
22     // do something - epilogue
23   }
24   b.recycle();
25 }

27 void foo_prologue(Point p, Object o1, Object o2, Node root, IntermediaryState interState) {
28   Object o3 = /* defined in prologue and used in epilogue */
29   Object o4 = /* defined in prologue but not used in epilogue */
30   Object pp = /* something dependent on p */
31   Object oo1 = /* something dependent on o1 */
32   Node root = /* something loop invariant */
33   // do something - prologue
34   interState.block.add(pp, o2);
35   interState.save_foo_o3(o3);
36 }

38 Object foo_epilogue(Point p, Object o1, Object o2, Node root, IntermediaryState interState) {
39   Object o3 = interState.load_foo_o3();
40   // do something - epilogue
41   return ret;
42 }
```

Fig. 4.8. Intermediary methods - transformed code



(a) Iteration space, original (b) Iteration space, post-blocking

Fig. 4.9. Iteration space of prologues and epilogues

was loop variant at the enclosing for loop. Hence `pp` is also considered loop variant (with regard to the enclosing for loop), and added to the block.

The actual `recurse` call is invoked at the level of the enclosing for loop. Loop *invariant* arguments to `recurse` may not be available at this level. Such arguments can be identified by following a chain of *transformed, loop invariant* intermediary methods up to the `recurse` call. These transformed methods have only statements which are loop invariant, and suffice to save loop invariant arguments `oo1` and `root` which are necessary to invoke `recurse`. Since any loop variant arguments will have been saved in the block, the combination comprises the entire `recurse` call. In our example `foo_map` is called with the loop invariant argument `o1` (line 8). This records the loop invariant arguments `oo1` and `root`, which are passed directly to `recurse` (line 16).

A graphical depiction of an iteration space with intermediary methods is shown in Figure 4.9 for a block size of 3. There are three intermediary methods, and the prologues are denoted P1-P3, epilogues E1-E3 and the recursion R. Originally, the order of execution is prologues, recursion, and epilogues, for each point. In the blocked iteration space, the prologue is executed for all points in the block, then the recursion is executed on the block. When the recursion returns, the epilogue is executed for all points in the block

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

Fig. 4.10. High level view of traversal splicing

4.4 Implementation of traversal splicing

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

Figure 4.11 is a template customized based on the optimization levels which is plugged in to the transformed code. Figure 4.13 is constructed by making a duplicate of the original recursive method, stripping all calls other than the *first* call of each call set, and adding code to save points and call set ids. Intermediary methods are split around the call path to the recursive method, into prologues and epilogues. Code is added to save any local state which persists from prologue to epilogue, and local variables within the recursive method which is passed as an argument to recursive calls. Autotuning code is added before any splicing setup is done (before line 3 of Figure 4.11), as the setup requires the splice depth, which is determined by the autotuner. Points consumed for autotuning are skipped in the point loop (lines 5-7).

4.4.1 A high level overview

Figure 4.10 shows a high level view of how traversal splicing is implemented. Recall that traversal splicing chops up a point's traversal into many partial traversals that are split at splice nodes, both explicit and implicit (Section 3.2.5). Bottom phases start at the children

of the explicit splice nodes (nodes at depth $D + 1$), but top phases start immediately after bottom phases complete. To find the nodes at which phases must begin, we map the top of the tree up to (and including) the children of splice nodes (line 3). Each of these nodes can be the start of a new top or bottom phase. For each of these nodes we allocate buffers into which points can be saved, so they can be resumed by top phases. We also allocate a buffer which can save $D + 1$ call set ids for each point (line 4).

The partial traversals are executed as pairs of top and bottom phases as described in Figure 3.6(a). Because a top phase can start at any node up to the splice depth, there are $2 \times p^D$ phases (where p is the maximum number of recursive calls within a call set, and 2 is for top and bottom each); `unrollRecursion` identifies each phase. Each top phase (line 6) executes a *single* path through the tree to an explicit splice node, saving points and call set ids into the buffers, so that traversals can be resumed by later phases. Each bottom phase (line 7) executes *all* paths of the subtree rooted at an explicit splice node, *without* saving points and call set ids. The last node visited in the bottom phase is by definition an implicit splice node.

The first top phase starts every point at the root of the tree. Subsequent top phases and bottom phases gather points saved into buffers from previous top phases, and resume them at appropriate nodes based on the call set id. Bottom phases resume only points paused at explicit splice nodes, whereas top phases also resume points saved above splice nodes due to truncation. Hence top phases gather points from *multiple* nodes, and dynamic sorting (described in Section 5.3.1) arises naturally as points are reordered based on the node they were saved at. There is no dynamic sorting at bottom phases, as they gather points from a *single* node.

4.4.2 Setup and recursion unrolling

In the actual implementation, the high level view of Figure 4.10 is moved to lines 1-8 of Figure 4.11. We map the top of the tree and save the nodes into an array based on a heap ordering of the tree as in Figure 2.3(a). Non-existent nodes are marked null in the array.

```
1 Set<Point> points = /* points */
2 Node root = buildTree(points);
3 mapTree(root);
4 allocBuffers();
5 foreach (Point p : points) {
6   recurseSplice(p, root, 0);
7 }
8 unrollRecursion(0, 0, 0, 0);

10 void allocBuffers() {
11   numLeafNodes = power(maxChildren, D + 1);
12   numNodes = numLeafNodes * maxChildren;
13   for (int i = 1; i < numNodes; i++) {
14     if (treeMap[i] != null) allocPointBuffers(i);
15   }
16   allocCallSetIdBuffer();
17 }

19 void unrollRecursion(int d1, int p1, int d2, int p2) {
20   if (d2 < D) {
21     for (int i = 0; i < maxPhases; i++) {
22       if (i == 0) unrollRecursion(d1, p1, d2 + 1, i);
23       else unrollRecursion(d2 + 1, i, d2 + 1, i);
24     }
25   } else {
26     if (d1 != 0) topPhase(d1, p1);
27     bottomPhase();
28   }
29 }
```

Fig. 4.11. Pseudocode of recursion unrolling

Point buffers into which points can be saved are allocated for all non-null nodes. Both the tree map and point buffers are indexed from 1 (the root) to $C^{D+2} - 1$ (the rightmost leaf), where C is `maxChildren`, and D is the splice depth. Point buffers need to accommodate an arbitrary number of points and are implemented as ArrayLists. The call set id buffer is a two dimensional array of size $P \times D$ (P is the number of points).

The partial traversals of all points are executed by pairs of top and bottom phases. While bottom phases are equivalent to normal recursive traversals, top phases execute a path

```
1 void topPhase(int depth, int phase) {
2   int start = power(maxChildren, depth);
3   int end = start * maxChildren;
4   for (int i = start; i < numNodes; i++) {
5     swapPointBuffersAndClearDest(i);
6   }
7   for (int i = start; i < end; i++) {
8     int substart = i;
9     int subend = i + 1;
10    while (substart < numNodes) {
11      for (int j = substart; j < subend; j++) {
12        foreach(Point p : getPointsAtNodeIndex(j)) {
13          Node n = nextNode(p.getCallSet(depth), phase);
14          recurseSplice(p, n, depth);
15        }
16      }
17      substart *= maxChildren;
18      subend *= maxChildren;
19    }
20  }
21 }

23 void bottomPhase() {
24   for (int i = numLeafNodes; i < numNodes; i++) {
25     foreach(Point p : getPointsAtNodeIndex(i)) {
26       for (int j = 0; j < maxPhases; j++) {
27         Node n = nextNode(p.getCallSet(D), j);
28         recurse(p, n);
29       }
30     }
31   }
32 }
```

Fig. 4.12. Pseudocode of top and bottom phases

through the tree down to the splice depth, and needs to save additional state so that traversals can be resumed. Figure 4.13 shows a *transformed* recursive method, `recurseSplice`, to realize top phases for the pseudocode for the nearest neighbor code of Figure 3.5. `recurseSplice` performs only the *first* traversal of each call set, later traversals in the call sets will be directly called (on the appropriate child) in subsequent phases. Points that

```
1 void recurseSplice(Point p, Node n, int depth) {
2   if (!canBeCloser(p, n.boundingBox)) {
3     n.savePoint(p);
4     return;
5   } else if (n.isLeaf()) {
6     p.updateClosest(n.getPoint());
7     n.savePoint(p);
8   } else {
9     double split = p.value(n.splitType);
10    if (split <= n.splitValue) {
11      if (depth < D) {
12        recurseSplice(p, n.leftChild, depth + 1);
13      } else {
14        n.leftChild.savePoint(p);
15      }
16      p.saveCallSet(depth, 0); // call set 0
17    } else {
18      if (depth < D) {
19        recurseSplice(p, n.rightChild, depth + 1);
20      } else {
21        n.rightChild.savePoint(p);
22      }
23      p.saveCallSet(depth, 1); // call set 1
24    }
25  }
26 }
```

Fig. 4.13. Transformed recursive method for top phases

reach the splice depth are saved into the node at which they should be *resumed* (not the node at which the point is paused) (lines 14 and 21), and points that are truncated beforehand are saved into the node at which they are truncated (lines 3 and 7). The call set id is saved per point per depth (lines 16 and 23).

Because the transformed code performs only the *first* traversal of a call set, subsequent partial traversals of the tree above the splice nodes are performed by later top phases. Top phase traversals are paused at explicit splice nodes, and the traversal is resumed by bottom phases that continue the traversal to the implicit splice node. The order of phases can be determined by partially unrolling the recursive traversal (`unrollRecursion` in lines 19-

29 in Figure 4.11), which basically expands the remaining top phases and appends a bottom phase after each top phase. The first top phase is started by calling `recurseSplice` for all points (line 6 of Figure 4.11. The depth of the nodes at which to resume and the phase number are passed as arguments to the top phase. For a bottom phase, the depth argument will always be $D + 1$, and all phase numbers are iterated upon. For our running example (Figures 3.4 and 3.6(a), $D = 2$, root is depth 0), the order of phases is T0-0 (top phase at depth 0, traversal phase 0), B (bottom phase), T2-1, B, T1-1, B, T2-1, B (as in Figure 3.6(a)). The first top phase always performs traversal phase 0 (the first traversal), and because our example has two phases per call set, all other top phases start with the second traversal in the set.

4.4.3 Dynamic sorting

In our implementation the dynamic sorting comes naturally, as points are reordered during top phases based on the node they were saved at. At each top phase, we gather all points which should execute in this phase (*i.e.*, all points that have not been truncated *above* the level of the top phase), and resume them at the appropriate node based on the point's call set id (lines 37-50 of Figure 4.11). For example at T2-1, we will gather points that have been paused or truncated at nodes ④–⑮, and resume at nodes ④–⑦. At T1-1 we will gather points at nodes ②–⑮, and resume at nodes ② and ③.

More specifically for our particular example of NN, at T2-1, we will gather points at nodes ④, ⑧, ⑨ and resume at node ⑤, because these points have visited {leftChild} and should now visit {rightChild}. The points which resume at ⑤ are better sorted now because they are in order of truncation at ④, reach ⑧ first, and reach ⑨ first. Similarly, points at nodes ⑤, ⑩, ⑪ resume at node ④, points at nodes ⑥, ⑫, ⑬ resume at node ⑦, and so on.

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

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

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

4.4.4 Optimizations

For inferred order algorithms, as in nearest neighbor, the call set id is unnecessary, and the next node can be determined completely by the previous node the point was at, and the phase number. Hence `i` can replace `p.getCallSet()` (lines 43 and 57 in Figure 4.11), and loop invariant statements can be hoisted out of the loop.

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

Splice node elision can be realized by executing a *merged* bottom phase at line 23 of Figure 4.11, if the depth is a fixed distance from D. The merged bottom phase uses the code for `topPhase` but doesn't swap point buffers and calls `recurse` instead of `recurseSplice`.

4.4.5 Local variables and intermediary methods

Because traversal splicing entails chopping up the recursion, it also requires that all methods surrounding the recursive method be split into prologues and epilogues. All prologues are executed before the first top phase, and all epilogues are executed after the last bottom phase. This requires that any local variables which are defined in the prologue(s)

and used in the epilogue(s) be saved in additional space allocated per point. Further, any local variables within the recursive method passed as an argument to subsequent recursive calls must also be saved for all points at all depths. This can contribute significantly to heap usage as shown in Section 4.6.2.

4.4.6 Discussion

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

4.5 Autotuning

In the previous section, we have presented an approach to identify recursive structures in tree traversal algorithms that can be transformed correctly, and a systematic method to transform the program to enhance locality. Critical to the performance of the transformation is the block size B and splice depth D. These parameters are dependent on both machine parameters (*e.g.*, L1, L2 cache size) and algorithmic characteristics (*e.g.*, density of a block at different depths, average traversal size). The notion of optimization parameters affecting the performance of transformations is well known. In recent years, there has been a large amount of research on *autotuning*, where a compiler automatically selects the best optimization parameters for a particular scenario [28–30]. In this section, we describe autotuning methods that automatically select a good block size to use for the transformed algorithm.

Many autotuners can operate at compile time. For example, in dense linear algebra, tile size is dependent on machine parameters but independent of input characteristics and hence can be determined when the library is compiled [30]. Because the optimal parameters for point blocking and traversal splicing are dependent on input characteristics as well as machine parameters, TREESPLICER's autotuners must operate at run-time, when the input is available.

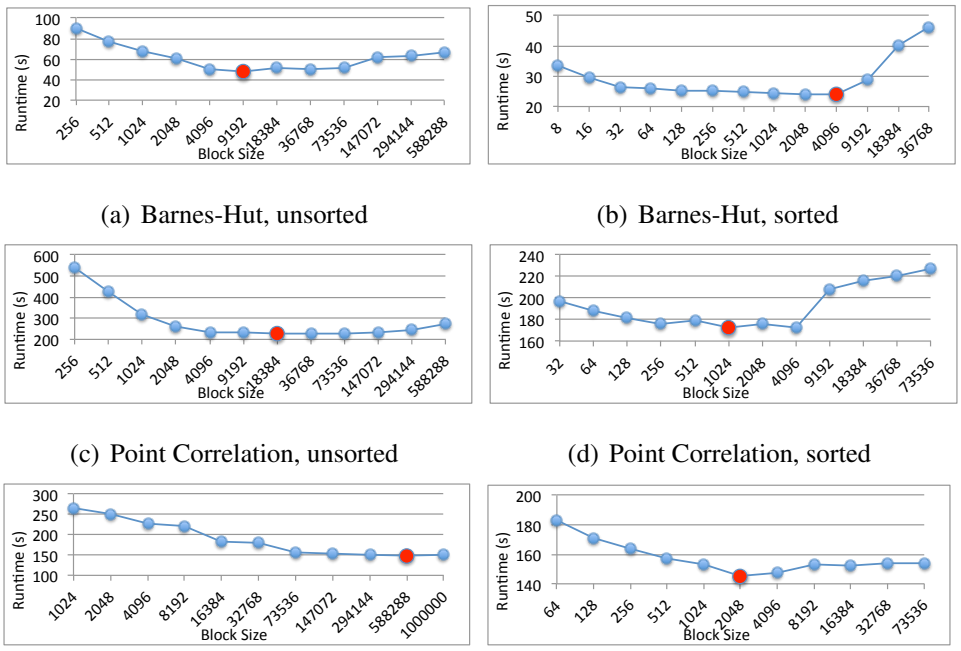
4.5.1 Tuning for point blocking

In order to autotune point blocking to choose the best size, we must have some idea of the behavior of different block sizes. Figure 4.14 shows the runtimes of point blocking with varying block sizes for three benchmarks, with and without the point sorting optimization (as described in Section 2.2)[3]. Intuitively, we would expect that a block size that is too small would perform poorly due both to the additional instruction overhead incurred by point blocking and to the fact that misses in the tree are incurred for every block (as discussed in Section 3.1)—more blocks will result in more misses in the traversal. However, if the block becomes too large to fit in cache, then we will begin to incur misses on the points instead. We thus expect there to be a "sweet spot," where the blocks are large enough to avoid most misses in the tree, but small enough to fit in cache, an expectation borne out by the results. In each figure, the best block size is highlighted, and is surrounded by block sizes that perform worse. The best block size is larger when the point sorting optimization is not performed, because points' traversals diverge, and the *effective* block size becomes small quickly.

We adopt a "guess and check" approach for autotuning, where a small portion of the points are consumed to test various block sizes, and the best block size is used for the remaining points. There should be a limit on the number of points used for autotuning to keep its overhead from becoming too high. We set the limit to up to 5% of the total points[4]. We start with a block size of 8, and run each block size 5 times to average out irregularities.

[3]These results are with the random inputs on the Opteron I system described in Section 4.6.1

[4]We found a 5% autotuning portion to deliver better performance than 1% as used in prior work [6].



(a) Barnes-Hut, unsorted

(b) Barnes-Hut, sorted

(c) Point Correlation, unsorted

(d) Point Correlation, sorted

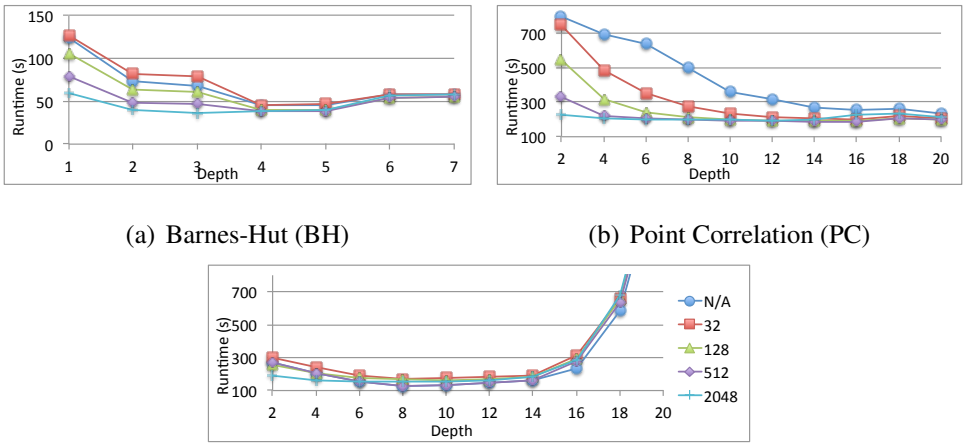
(e) Nearest Neighbor, unsorted

(f) Nearest Neighbor, sorted

Fig. 4.14. Runtime of point blocking with varying block sizes

For one million points, this allows us to test up to a block size of 4096. We also test the base case (the original code path), to check if we should be applying our transformation at all.

Complicating matters, the irregular nature of the algorithms means that different regions of the data structure might exhibit widely differing characteristics (*e.g.*, the traversals of points that walk one part of a tree might be much shorter than those that walk a different part of the tree). An autotuner that investigates various block sizes only using points from early in the execution may not see the full range of possible behaviors. A common approach to account for this variability is to use random sampling. Randomly selecting the test points from among all the points provides on average the best representative of the entire set attainable. Because the points may have been sorted so that consecutive points have similar traversals and enjoy temporal locality, we would like to take random samples of



(a) Barnes-Hut (BH)

(b) Point Correlation (PC)

(c) Nearest Neighbor (NN)

Fig. 4.15. Runtime with varying block sizes and splice depths

blocks. When testing a block size of 8, we choose a random sequence of 8 consecutive points, rather than constructing the block from 8 randomly sampled points

The irregular nature also means that different points (and hence different blocks) will do different amounts of work (*i.e.,* traverse different numbers of nodes). We normalize recorded runtimes to the actual work done for a fair comparison. The actual work done is profiled by recording the block size at the start of each recursive method call. The autotuning phase executes a different code path, so that this profiling overhead is not incurred during the main phase.

An important correctness condition is that sampled points must be skipped in the point loop after the autotuning phase. This is because processing a point can have side-effects, and processing a point twice can be incorrect. The transformed code maintains a boolean array to mark sampled points, and skips sampled points in the main loop.

4.5.2 Tuning for traversal splicing

Critical to the performance of splicing is selecting a good splice depth, D. For point blocking, we proposed an autotuning technique for selecting block size where a small portion of the input points are used to construct blocks of varying sizes, and the best performing

block size is chosen as the transformation parameter. A similar approach is infeasible for splicing, because we would like to apply splicing over the full set of points. In principle, we would like to choose the splice depth, D, such that each partial traversal completely fits in cache. In practice, because tree topology is unpredictable, and the size of partial traversals is input and algorithm dependent, finding a suitable D is difficult.

Recall that in Section 3.2.2, we observed that point blocking can be applied on top of traversal splicing to mitigate the effects of large partial traversals. Figure 4.15 shows the runtime of three benchmarks with varying block sizes and splice depths (on the Opteron system described in Section 4.6.1). The five lines compare splicing only, to splicing with blocking applied to the bottom phases[5] with block size 32–1024. We note that point blocking reduces traversal splicing's sensitivity to D for BH and PC, which have large bottom phases, and does not harm the performance of NN, which has small bottom phases. Serendipitously, this means that by combining splicing and blocking, we need not be as careful about choosing D. Empirical results suggest that a depth at half of the average *reach* (the depth of the nodes where a point's recursion is stopped) is a good splice depth.

We record the average reach of all points in the test blocks. If we are not applying blocking, we use 10 randomly selected points to compute the reach. We set D as half of the average reach. We also determined empirically that splice node elision (see Section 3.2.5) should be performed if a top phase begins fewer than $D/2$ levels above D.

4.5.3 Autotuned parameters

Table 4.1 shows the autotuned parameters averaged (and rounded to the first digit) across the recorded serial runs on the benchmarks and systems described in Section 6.2. Block sizes are shown for point blocking without sorting (**Block**) and with sorting (**Sort+Block**). The autotuned block sizes when traversal splicing is combined are similar. The maximum block size is limited to 0.5% of the total points, so that each block size can be tested 5 times, and the autotuning overhead limited to 5% of the points. When points are not sorted,

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

Table 4.1
Autotuned parameters

Benchmark	Input	Block Sizes								Splice Depths			
		Block				Sort+Block				Splice			
		Opt I	Opt II	Opt III	Xeon	Opt I	Opt II	Opt III	Xeon	Opt I	Opt II	Opt III	Xeon
BH	Random	4096	4096	4096	4096	512	853	427	427	4	4	4	4
	Plummer	4096	4096	4096	4096	256	256	256	512	5	5	5	5
PC	Random	4096	4096	4096	4096	683	512	4096	1877	10	11	11	10
	Covtype	512	512	512	512	512	512	512	213	9	9	9	9
	Mnist	512	512	512	512	512	512	512	512	9	9	9	9
NN	Random	4096	4096	4096	4096	3413	4096	4096	4096	9	9	9	9
	Covtype	512	512	512	512	512	512	427	512	9	9	9	9
	Mnist	512	512	512	512	512	512	512	512	9	9	9	9
kNN	Random	4096	4096	0	4096	4096	4096	4096	4096	10	10	10	10
	Covtype	512	512	0	512	512	512	427	512	9	9	9	9
	Mnist	512	512	0	512	512	512	512	512	9	9	9	9
BT	Random	2048	2048	2048	2048	2048	2048	2048	2048	8	8	8	8
	Covtype	512	512	512	512	512	512	512	512	8	8	8	8
	Mnist	512	512	512	512	512	512	427	512	8	8	8	8
RT	Random	8192	8192	8192	8192	6827	8192	5461	6827	10	10	10	10

the selected block sizes are often capped at the maximum (4096 for random inputs with one million points, and 1024 for real inputs with $200,000$ points, 8192 for RT with 2^{23} points), because the blocks become sparse quickly. For kNN the divergence is worse enough that sometimes the autotuner decides not to do point blocking. The selected block sizes are smaller when sorting is performed.

Splice depths are shown for pure traversal splicing (**Splice**). The autotuned splice depths when point blocking is combined are similar. Splice depths are determined by the average reach of points, and is machine independent and sorting independent (*i.e.,* cache size or point order does not change average reach). Slight differences are due to the random sampling of points for autotuning. BH has a smaller depth because it is an octtree. For other benchmarks, random inputs have larger depths than real inputs, because there are more points, and the tree is larger.

4.6 Evaluation

This section presents our experimental evaluation. We start with evaluation methodology, then explore the memory overheads of our transformations. Next we discuss ex-

perimental results for serial runs of each of our benchmarks, and examine the instruction overheads and locality benefits through performance counters. Finally we discuss parallel results and demonstrate that the improvements from our transformations scales to many threads.

4.6.1 Evaluation methodology

To demonstrate the efficacy of TREESPLICER, we evaluate it on six tree traversal algorithms, from various domains ranging from scientific applications to data-mining and graphics.

- **Base**: the baseline described for each benchmark below.
- **Block**: automatic point blocking as described in Sections 3.1 and 4.3.
- **Splice**: automatic traversal splicing as described in Sections 3.2 and 4.4.
- **Block+Splice**: automatic traversal splicing combined with automatic point blocking for bottom phases.

We add application-specific *a priori* point sorting to each above version, to get eight versions. Sorting is done in tree order [17] for all benchmarks other than RT. For RT, the unsorted baseline uses the default ordering, where shadow rays are processed in the order they are produced (*i.e.*, in the order of the initial eye rays), while the sorted version groups shadow rays according to their light source. Sorting time is included for NN, kNN and BT where sorting has additional overhead , amounting to between 0.3%–5.8% of the traversal time.

The benchmarks were written in Java and executed on the HotSpot VM 1.7 with 12GB heap. Each configuration was run 8 times in a single VM invocation, the first run and the min/max runs dropped, and the mean of 5 runs was recorded. We enforced a coefficient of variation[6] of 0.05 by extending the number of runs until steady state if necessary, and this yields errors of at most $\pm 6.21\%$ of the mean with 95% confidence [44].

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

Benchmarks

Barnes-Hut (BH) is a scientific kernel for performing n-body simulation [3]. All n bodies are placed into an oct-tree. Each body traverses the tree to compute the force(s) acting upon it. In the terminology of our optimization classes from Section 3.2.5, BH is an inferred order algorithm with a single call set. We use the implementation from the Lonestar benchmark suite [45] with a single iteration, and two inputs. **Random** is one million randomly generated bodies. **Plummer** is the class C input from the Lonestar suite, with one million bodies generated from a Plummer model.

Point Correlation (PC) is described in detail in Section 2. PC is an inferred order algorithm with a single call set. We use three inputs. **Random** has one million randomly generated points in 3 dimensions. **Covtype** is real data on forest cover type with $580{,}000$ points in a 54-dimensional space [46]. We reduced the dimensionality to 7 via random projections [47], and took $200{,}000$ points in random order. **Mnist** is real data on handwritten digits with $8{,}100{,}000$ points in a 784 dimensional space [48]. We again reduced the dimensionality to 7, and took $200{,}000$ points.

Nearest Neighbor (NN) is described in Section 3.2.3. This implementation saves a bounding box of all points within each node of the kd-tree, and splits nodes until there are four points or fewer in the leaf nodes. NN is an inferred order algorithm, but has *two* call sets. We use three inputs with separate training and test sets. For each point in the test set, we find the nearest neighbor in the training set. **Random** has a training set and test set of one million points each, randomly generated in a 7-dimensional space. We also used the **Covtype** and **Mnist** inputs described above, with a training set and test set of $200,000$ points each.

k-Nearest Neighbor (kNN) is an optimized k-nearest neighbors benchmark, using a different kd-tree variant than NN. This implementation does not save a bounding box per node. Instead it uses the difference between the split plane and the query point's corre-

sponding dimension value as a termination condition. This condition is less aggressive in pruning nodes than the bounding box, but requires less computation. This implementation also saves the median point in non-leaf nodes, providing further speedup by reducing the number of nodes traversed. kNN is an inferred order algorithm with two call sets. We use the same three inputs as for NN, with $k = 5$.

Ball Tree (BT) is a nearest neighbor benchmark using a ball tree [49]. We use the implementation from the WEKA data mining software [50]. For efficiency reasons, our baseline uses an optimized distance computation that is tailored to our inputs. However, we did not alter the data structure or the actual traversal code; TREESPLICER was applied to the "out of the box" traversal algorithm. The distance computation kernel we modified is not touched by our transformations. BT is an inferred order algorithm with two call sets. We use the same three input sets as for NN, with **Random** having $600,000$ instead of one million points.

Ray Tracing (RT) can be accelerated with tree-structured bounding volume hierarchies (BVHs) that accelerate ray-object intersection tests. Our benchmark is extracted from the BVH-based ray tracer of Walter *et al.* [40]. Ray tracing is the most general benchmark we tackle, as it is *not* an inferred order algorithm and has *four* call sets, and hence we must track each call set explicitly[7]. However, it is pseudo-tail recursive. The input is a randomly generated scene with four million triangles. We rendered a single frame with 512×512 eye rays and 32 lights (hence, 32 shadow rays per eye ray).

Platforms

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

- **Opteron I** runs Linux 2.6.24 and contains two dual-core AMD Opteron 2222 chips. Each chip has 128K L1 data cache per core and 1M L2 cache per core.

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

Table 4.2
Transform time, traversal time and heap usage

Benchmark	Lines of code	Transform time (ms)	Input	Traversal time %	Traversal time (s)	Heap usage			
						Base (MB)	Splice (% increase)		
							-O0	-O1	-O2
Barnes-Hut	466	1045	Random	85.3	222.3	210	127.6	46.7	18.6
			Plummer	90.5	427.8	210	90.5	31.4	18.1
Point Correlation	396	1037	Random	99.4	791.1	188	132.4	21.3	2.1
			Covtype	98.9	885.8	59	96.6	18.6	8.5
			Mnist	95.3	221.9	56	107.1	21.4	5.4
Nearest Neighbor	450	1004	Random	98.0	314.9	390	47.2	2.6	
			Covtype	96.5	467.1	72	70.8	9.7	
			Mnist	96.6	593.1	74	60.8	2.7	
k-Nearest Neighbor	378	1114	Random	99.9	739.8	478	72.8	59.4	
			Covtype	88.2	117.5	92	78.3	52.2	
			Mnist	93.3	268.4	92	72.8	50.0	
Ball Tree	6199	2060	Random	98.6	969.1	316	51.3	39.2	
			Covtype	92.8	268.0	84	90.5	76.2	
			Mnist	96.8	741.5	84	92.9	76.2	
Ray Tracing	3988	1960	Random	55.1	251.5	781	159.9		

- **Opteron II** runs Linux 2.6.32 and contains four twelve-core AMD Opteron 6176 chips. Each chip has 64K L1 data cache per core, 512K L2 cache per core, and two 6M shared L3 caches.

- **Opteron III** runs Linux 2.6.32 and contains four sixteen-core AMD Opteron 6282 chips. Each chip has 16K L1 data cache per core, 2M L2 cache per two cores, and a 16M shared L3 cache.

- **Xeon** runs Linux 2.6.32 and contains four eight-core Intel Xeon E5-4650 chips. Each chip has 32K L1 data cache per core, 256K L2 cache per core, and a 20M shared L3 cache.

Traversal times

For each of our benchmarks, we measure the amount of time spent in traversals, as this is the portion of code our transformations change. RT has two traversal phases, the first where initial rays are cast, and the second where shadow rays are traced. Because the initial rays in the first phase are sorted inherently, we measure the second phase to demonstrate the efficacy traversal splicing on unsorted points[8]. Column 5 of Table 4.2 shows the percentage of total time spent in traversals for each baseline benchmark/input on the Opteron I system. It can be seen that the traversal time is the dominant component of the total runtime. The exception in RT is because the BVH tree takes considerable time to build. We note that for real applications the same BVH will be used for multiple frames, making the traversal time dominant. Hereafter we will discuss the performance only of the traversal portions of the benchmarks.

Transformation times

Columns 2–3 of Table 4.2 shows the lines of code and transformation times for our benchmarks[9]. Our transformations are very quick, amounting to less than two seconds.

4.6.2 Heap usage and optimizations

Both point blocking and traversal splicing process multiple points simultaneously and require additional space to save state of paused points. This additional state overhead is proportional to the block size for point blocking, and is insignificant as the block size is much smaller than the total number of points. The overhead can be significant for traversal splicing, as *all points* are in-flight simultaneously. Columns 7–10 of Table 4.2 shows the heap usage of the baseline algorithms, and the percentage increase of the transformed, spliced implementations (with splice node elision) at three optimization levels:

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

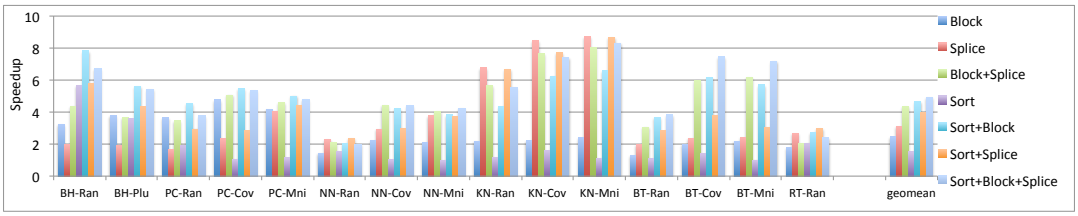
[9]Transformation times are on an Intel Core 2 Duo with blocking, splicing, and all optimizations applied.

- **-O0** is the default spliced implementation for pseudo-tail recursive codes (Section 3.2.5).
- **-O1** is an optimized spliced implementation which exploits inferred order (Section 3.2.5).
- **-O2** is an optimized spliced implementation for codes with a single call set (Section 4.4.4).

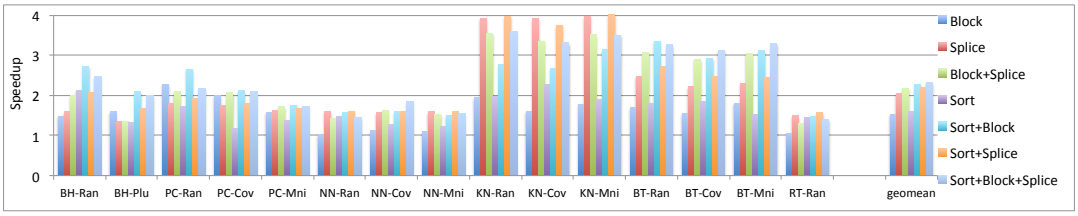
The percentage increase is left blank when the optimization level is not applicable to the algorithm. The memory overheads of splicing comes from extra state used to save four types of data: (i) paused points; (ii) call set ids for non-node-inferred-order algorithms; (iii) local variables in intermediary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

Table 4.3
Geometric mean of serial speedups

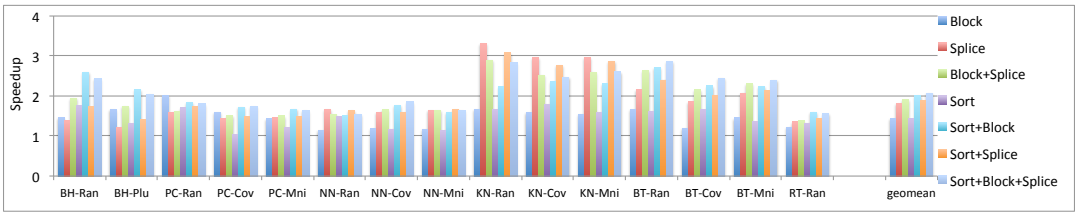
System	Cache size			Mean speedup						
	L1	L2	L3	**Block**	**Splice**	**Block+Splice**	**Sort**	**Sort+Block**	**Sort+Splice**	**Sort+Block+Splice**
Opteron I	128K	1M		2.45	3.09	4.35	1.51	4.67	4.00	4.89
Opteron II	64K	512K	6M	1.53	2.06	2.16	1.60	2.27	2.19	2.33
Opteron III	16K	2M	16M	1.44	1.82	1.92	1.43	2.00	1.89	2.07
Xeon	32K	256K	20M	1.55	2.05	2.08	1.78	2.16	2.14	2.20
Mean				1.70	2.21	2.48	1.57	2.60	2.44	2.68



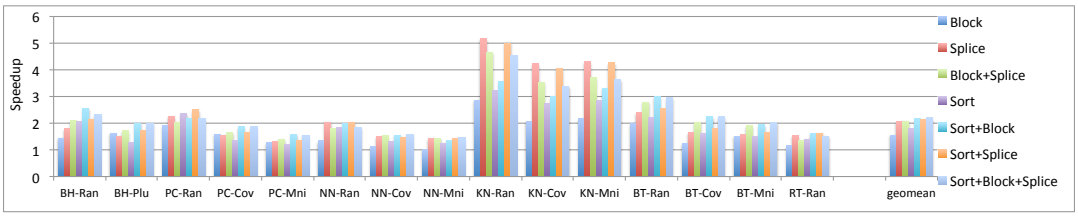
(a) Opteron I



(b) Opteron II



(c) Opteron III

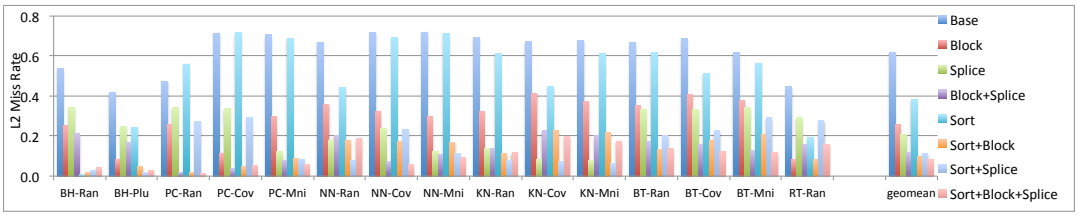


(d) Xeon

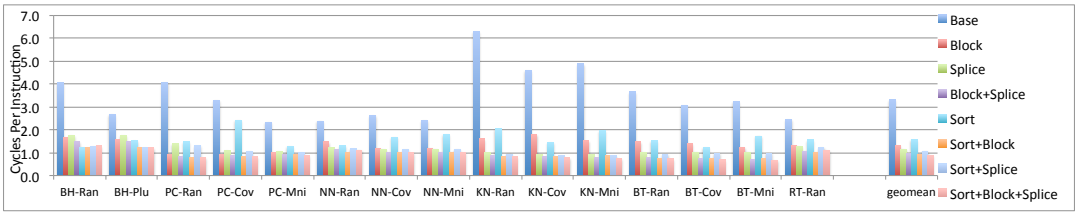
Fig. 4.16. Serial improvements: Speedup over **Base**

4.6.3 Serial results

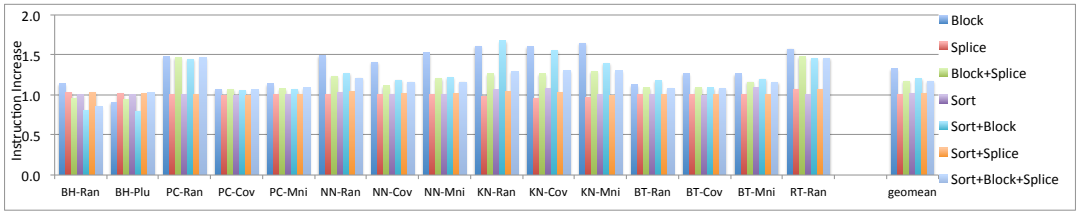
Figure 6.3 show speedups of **Block**, **Splice**, **Block+Splice**, **Sort**, **Sort+Block**, **Sort+Splice** and **Sort+Block+Splice** over **Base** for each benchmark/input pair, for the four systemsThe geometric mean of the speedups across all benchmark/input pairs on each system and across all four systems are summarized in Table 4.3. The speedups are markedly larger on Opteron



(a) L2 miss rate



(b) CPI (Cycle per instruction)



(c) Instruction counts normalized to **Base**

Fig. 4.17. Performance counters on Opteron II

I which has the smallest cache. A smaller cache results in the baseline having poor locality to start with, and leaves more room for our transformations to enhance locality. Overall the speedups are 1.70, 2.21, 2.48 for **Block**, **Splice** and **Block+Splice** respectively. Adding *a priori* point sorting increases speedups to 1.57, 2.60, 2.44 and 2.68 for **Sort**, **Sort+Block**, **Sort+Splice** and **Sort+Block+Splice** respectively.

We see that point blocking is often an effective optimization, even without an *a priori* sorting pass (1.70 speedup), but performs much better when combined with point sorting (2.60 speedup), as discussed in Section 3.1.2. Traversal splicing is less dependent on sorting, due to its ability to reorder points on the fly (2.21 speedup without sorting, 2.44 speedup with sorting). Traversal splicing consistently outperforms point blocking, often

by significant amounts when the points are unsorted, and is on average as good as point blocking when the points are sorted.

The efficacy of manual, *a priori* sorting compared to traversal splicing varies with benchmark and system. Manual sorting is particularly effective for BH, whereas it is particularly ineffective for kNN. Overall a fully automatic transformation, **Block+Splice** at 2.48 speedup, is competitive with the best configuration including application specific manual sorting, **Sort+Block+Splice** at 2.68 speedup, hence in many cases manual sorting is no longer necessary.

4.6.4 Performance counters

Figure 4.17 shows performance counter results on the Opteron II collected with PAPI [51]. The geometric means are summarized in Table 4.4. The geometric mean of the L2 miss rate is 0.618 for **Base**. Applying point blocking in **Block** reduces this to 0.258. Applying traversal splicing in **Splice** independently brings the mean L2 miss rate to 0.208 and combining both point blocking and traversal splicing in **Block+Splice** reduces this to 0.117 which is less than a fifth of the baseline L2 miss rate. Further applying sorting brings the L2 miss rates down to 0.385, 0.098, 0.114 and 0.083 for **Sort**, **Sort+Block**, **Sort+Splice** and **Sort+Block+Splice** respectively.

Because the Opteron II has a large L3 cache, CPI (cycles per instruction) can be a more comprehensive indicator of locality. The enhanced locality is reflected in the CPI, which on average is 3.31 for **Base**, but is reduced to 1.32, 1.15 and 0.94 for **Block**, **Splice** and **Block+Splice** respectively. Further applying sorting brings the CPI down to 1.59, 0.91, 1.07 and 0.89 for **Sort**, **Sort+Block**, **Sort+Splice** and **Sort+Block+Splice** respectively.

Figure 6.4(a) shows the instruction increase of our transformations compared to **Base**. Point blocking often has substantial instruction overhead (1.33), because highly divergent points result in sparse blocks. This overhead is reduced when the points are sorted and the blocks are denser (1.20). The dynamic sorting of traversal splicing also makes the blocks denser, and reduces the instruction overhead of point blocking (1.17). Traversal splicing

Table 4.4
Geometric mean of performance counters on Opteron III

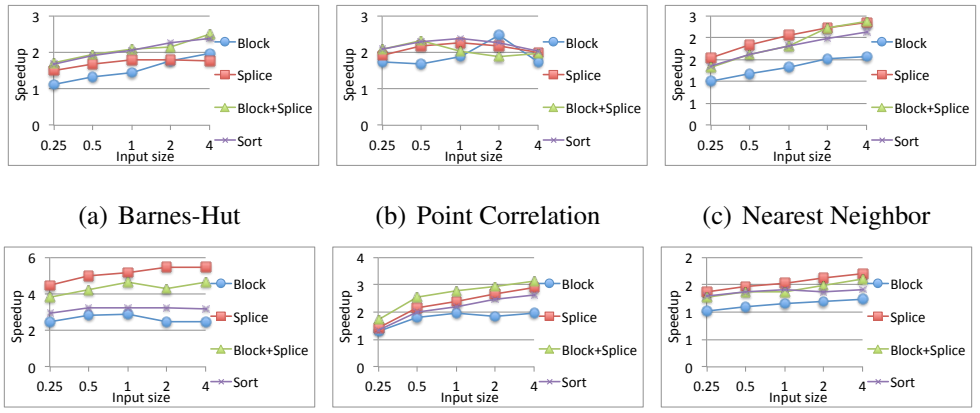
	Mean values							
	Base	**Block**	**Splice**	**Block+Splice**	**Sort**	**Sort+Block**	**Sort+Splice**	**Sort+Block+Splice**
L2 miss rate	0.618	0.258	0.208	0.117	0.385	0.098	0.114	0.083
CPI	3.31	1.32	1.15	0.94	1.59	0.91	1.07	0.89
Instruction increase		1.33	1.00	1.17	1.02	1.20	1.02	1.17

itself has negligible overhead. Overall these slight instruction increases are well justified by the enhanced locality of our transformations, resulting in the substantial speedups discussed in the previous section.

4.6.5 Input sizes

Figure 4.18 shows the speedup of **Block**, **Splice**, **Block+Splice** and **Sort** for varying input sizes of the **Random** input of each benchmark. 1 is the reference input size as discussed in Section 4.6.1, and the input size is varied from $4\times$ smaller to $4\times$ larger than the reference. We expect larger inputs to have more speedup, as the baseline suffers from worse locality, and there is more room for locality improvement. This expectation is borne out by the results.

Note that the increase in improvements as input sizes increase are gradual, whereas such graphs for regular programs (*e.g.,* matrix multiply) would have sharp drop-offs at cache size boundaries. This is due to the irregular nature of tree algorithms: larger input sizes gradually increase the likelihood that a traversal will outstep cache, instead of causing all traversals to outstep cache at a fixed threshold.



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

(d) k-Nearest Neighbor (e) Ball Tree (f) Ray Tracing

Fig. 4.18. Speedup for different input sizes on Xeon

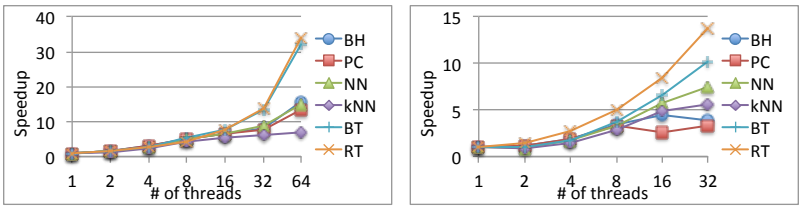
4.6.6 Parallel improvements

TREESPLICER takes sequential code as input, and outputs sequential code[10]. To test our benchmarks on multicores, we manually parallelized each benchmark, with Java threads. Each version was parallelized by statically and uniformly distributing the points among threads. We present results up to 64 threads for the Opteron III, and up to 32 threads on the Xeon. In the interests of brevity, we present results for the random inputs for all benchmarks.

We first present the scalability of the benchmark baselines in Figures 4.19(a) and 4.19(b), to show that the baselines do indeed scale. Due to lack of space we show parallel improvements for four versions: **Splice** and **Block+Splice** as fully automatic transformations, **Sort** as prior work exploiting semantic information, and **Sort+Block** as a combination of automatic transformations and semantic information. Figures 4.20 and 4.21 show parallel improvements of these versions over **Base** for each benchmark.

Locality is important for parallel scalability because multicores have limited bus bandwidth, and too many cache misses can saturate the bus. Enhanced locality means fewer

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



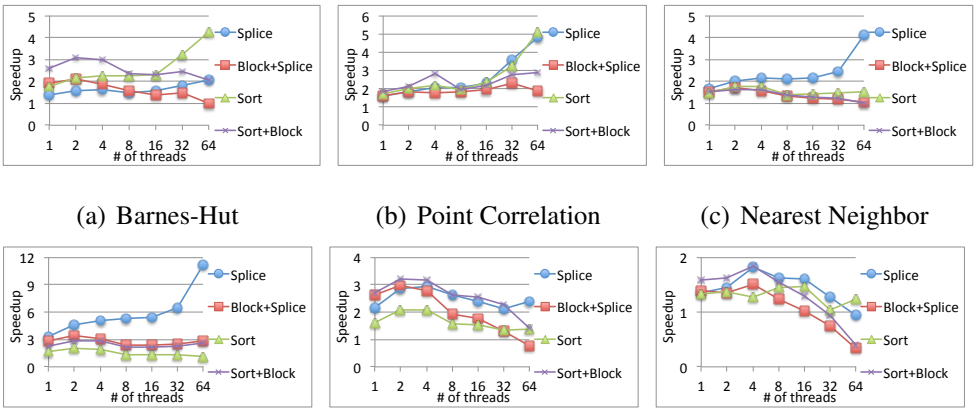
(a) Scalability on Opteron III (b) Scalability on Xeon

Fig. 4.19. Scalability of **Base**

cache misses and reduced bus pressure. Hence the optimized versions can perform more operations before saturating the bus. We expect that the optimized versions will perform better compared to the baseline when the baseline is saturated, but the optimized versions are not. However eventually even the optimized versions can become saturated, at which point the relative improvement declines.

There are two factors which diminish the improvements of our transformations at scale. First, autotuning is currently done sequentially and limits speedup according to Amdahl's law. While we have limited the autotuning portion of point blocking to 5% for the serial case, parallelization can increase its overhead to up to 77% for 64 threads. This overhead reduces the improvement of **Block+Splice** and **Sort+Block** as the number of threads increase. Autotuning for traversal splicing uses 10 points to compute the average reach, and its overhead is negligible. Second, splicing's dependence on exploring large numbers of points to exploit reordering means that its relative improvement drops as scale increases, as each thread processes fewer points. This effect should be mitigated for larger inputs, where each thread will receive more points.

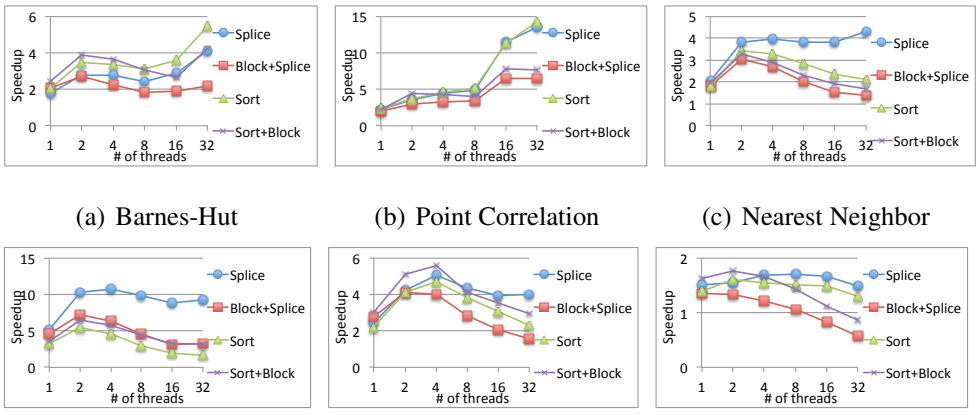
These trends combined can be seen in the results. The efficacy of **Sort** varies per benchmark, but **Splice** consistently delivers good performance at all scales, often significantly larger improvements at 32–64 threads because the improved locality reduces bus pressure, whereas the baseline is bus saturated.



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

(d) k-Nearest Neighbor (e) Ball Tree (f) Ray Tracing

Fig. 4.20. Speedup of transformed versions on n threads over **Base** on n threads on Opteron III



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

(d) k-Nearest Neighbor (e) Ball Tree (f) Ray Tracing

Fig. 4.21. Speedup of transformed versions on n threads over **Base** on n threads on Xeon

5. VECTORIZATION OF TREE TRAVERSAL ALGORITHMS

5.1 Scheduling transformations

In Section 3, we showed that tree algorithms have a common structure, and that this structure, despite its seeming irregularity, nevertheless enabled general *scheduling* transformations that rearrange accesses and improve locality. We developed a pair of transformations, *point blocking* [6] and *traversal splicing* [7], that can be applied to irregular tree-traversal algorithms to promote locality of reference. We explained these transformations in terms of loop tiling, a traditional locality optimization for regular programs that operate over dense arrays and matrices. Here, we present an alternate interpretation of these transformations that makes their applicability to SIMDization more clear.

In the abstract, we can think of each traversal in a traversal algorithm as being executed by a separate thread. The problem we must tackle is how to *schedule* the execution of these threads on a single processor. The original schedule of execution runs an entire thread (traversal) to completion before switching execution to the next thread. There are no context switches between threads. Even this simple schedule gives us some flexibility: after a thread finishes executing, we can choose any of the remaining threads to execute next. If we choose a thread whose traversal is likely to touch similar portions of the tree as the just-finished traversal, those parts of the tree are likely to still be in cache, improving locality. This is equivalent to *sorting* the points before execution so that points with similar traversals will be executed consecutively [17, 22].

5.1.1 Point blocking

When a thread's traversal gets too large to fit in cache, point sorting no longer suffices; even if the next thread's traversal is exactly the same, the nodes of the traversal will already

have been evicted from cache. We proposed point blocking to ameliorate this effect [6]. Intuitively, point blocking (and traversal splicing, as we shall see next) no longer executes a single thread to completion before running the next thread. Instead, point blocking context switches between threads to improve locality in the tree.

In point blocking, threads are grouped into blocks. A thread visits a single node of the tree (*i.e.*, it executes the body of the recursive method), but immediately after recursing down a child, control is transferred to another thread in the block that is at the same tree node, where the process repeats. If a thread in the block does not visit a particular tree node, its turn is "skipped." It will next execute when the threads in the block return to next tree node that its traversal would visit. Thus, all of the threads that must access a particular tree node access that node consecutively, giving good locality in the tree regardless of the size of the threads' traversals.

Constant context switches to achieve this scheduling would have high overhead. Hence, point blocking implements the schedule statically: points are grouped into blocks, and entire blocks traverse the tree, visiting a node if *any* of the points in the block want to visit it. Figure 3.2 showed how the code of Figure 2.1 is transformed to implement point blocking.

We note three salient facts. First, the transformation is straightforward: all returns in the main body of the loop are replaced with continues, and points that continue recursion are deferred until all the points have processed the current node. Second, if no points want to continue recursion, the "next" block is empty, so recursion stops; the block traverses the union of the sets of tree nodes that would have been visited by its constituent points. Finally, and most importantly, the main body of the recursive function now contains a simple for loop over a dense block of points (line 9). Each iteration of that loop performs the same set of instructions (disregarding any truncation), and the memory accesses within the loop are predictable: there is no pointer chasing involved. As Section 5.2 discusses, point blocking naturally enables SIMD execution.

5.1.2 Traversal splicing

Traversal splicing is an alternate approach to scheduling traversal codes to improve locality [7]. Intuitively, while point blocking interleaves traversals at the granularity of individual tree nodes, context switching between threads in a block at every instance of the recursive function, traversal splicing performs its context switching at a coarser granularity. In traversal splicing, various nodes in the tree are marked as *splice nodes*. A thread executes until its traversal reaches one of the splice nodes, or until its traversal truncates at a node that is an ancestor of the splice node. Then control is transferred to another thread, which similarly executes until it reaches the splice node. This process continues until *all* threads in the program have been "paused" at the splice node or truncated. Then control transfers back to the first thread, and execution continues until the next splice node, and so on. Hence, threads are interleaved at the granularity of "partial traversals" that traverse *between* splice nodes.

Though traversal splicing is more complicated than point blocking, it has a few advantages. Context switches in point blocking can only switch between threads in the same block of points. This restriction is good for locality: if control could switch between arbitrary threads, then thread-local (*i.e.*, point-specific) data may fall out of cache. However, it means the schedule is less likely to find points that will visit the same node in the tree. Because traversal splicing context switches between *all* threads in the traversal algorithm, if *any* threads visit the same node, they will be found and their accesses to the node will happen in close succession. Traversal splicing can also be combined with point blocking: while executing a partial traversal, multiple threads can be blocked and interleaved according to the point blocking strategy.

In addition to the locality-enhancing scheduling described above, traversal splicing allows a schedule to be *dynamically* tuned. When context switching between threads, it is useful if the next thread that is executed will have similar behavior as the previous thread (recall the purpose of sorting the points). Because threads are interleaved at the granularity of splice nodes, the scheduler can take the opportunity, at each splice node, to adjust the

interleaving order based on the previous history of each thread's execution. Threads that have behaved similarly up until a particular splice node are likely to behave similarly in the future, and can hence be reordered to execute consecutively. This technique is called *dynamic sorting*.

Interestingly, our prior investigations found that dynamic sorting does not improve locality significantly; because partial traversals tend to be small, the precise order of threads within those traversals tends not to matter. In this paper, we capitalize on a key insight: while dynamic sorting may not improve locality, it does improve the similarity of consecutive threads' traversals. As Section 5.3 elaborates, this can dramatically improve a traversal algorithm's SIMD potential.

5.2 Point blocking to enable SIMD

As described in Section 5.1.1, applying point blocking exposes a simple for loop over the block of points within the recursive function. Each iteration of that loop performs the same set of instructions, exposing an opportunity for SIMD execution. Point blocking with a block size of S (where S is the SIMD width) is directly analogous to packet-based SIMD approaches [5, 32]. A point block, or packet, traverses the tree, and the algorithm moves on to the next packet after the traversal of the previous packet is complete.

Packet-based approaches break down when the points in the packet behave differently. For example in Nearest Neighbor (NN) each point traverses either the left child or right child of a tree node first based on properties of the point and the node. Hence, traversals *diverge*, and a packet of S points can quickly degenerate into a single point after a few steps, in which case SIMD execution is no longer beneficial. This effect can be ameliorated by having *multiple* packets traverse the tree simultaneously, and compacting packets at each level to repopulate sparse packets and increase SIMD utilization.

While multi-packet traversal and compaction sounds like a complicated optimization, a simple modification of point blocking achieves this goal. Rather than applying point-blocking with a block size of S, we can apply point blocking with a larger block size, and

```
1  void processPoint(Block block,
2      Block nextBlock, int bi, TreeNode n) {
3    Point p = block.p[bi];
4    if (truncate(p, n)) {
5      updatePoint(p, n);
6      return;
7    }
8    nextBlock.add(p);
9  }

11 void recurse(Block block, TreeNode n) {
12   Block nextBlock;
13   int si = 0;
14   for (; si < block.size - S + 1; si += S) {
15     for (int bi = si; bi < si + S; ++bi) {
16       processPoint(block, nextBlock, bi, n);
17     }
18   }
19   for (int bi = si; bi < block.size; ++bi) {
20     processPoint(block, nextBlock, bi, n);
21   }
22   if (!nextBlock.isEmpty()) {
23     recurse(p, n.child1);
24     recurse(p, n.child2);
25   }
26 }
```

Fig. 5.1. Stripmined point blocking

then *strip mine* the resulting loop, adding an inner loop of S iterations. This inner loop can be SIMDized as a single packet, while at each step of the block's traversal, any non-truncated points can be compacted into full packets at the next level. The compaction is implemented by copying points from the current block into a "next block" that visits the children; points in this next block will be contiguous.

Figure 5.2 shows the result of SIMDizing the strip-mined loop, with $S = 4$. Note that in place of the inner (strip-mined) loop, we use a call to to `processVec` (line 17) that operates on the S points within the packet simultaneously with SIMD instructions (line 1). Within `processVec`, control flow is converted into masking operations represented by

```
1 void processVec(Block block, Block
2    nextBlock, int si, TreeNode n, int mask) {
3   __m128 vec_valid;
4   truncateVec(block, si, n, vec_valid);
5   updatePointVec(block, si, n, vec_valid);
6   mask = _mm_movemask_ps(vec_valid) & mask;
7   if (mask & 0x1) nextBlock.add(block, si);
8   if (mask & 0x2) nextBlock.add(block, si + 1);
9   if (mask & 0x4) nextBlock.add(block, si + 2);
10  if (mask & 0x8) nextBlock.add(block, si + 3);
11 }

13 void recurse(Block block, TreeNode n) {
14  Block nextBlock;
15  int si = 0;
16  for (; si < block.size - S + 1; si += S) {
17    processVec(block, nextBlock, si, n, 0xf);
18  }
19  int mask = getValidMask(si, block.size);
20  if (mask > 1) {
21    processVec(block, nextBlock, si, n, mask);
22  } else if (mask == 1) {
23    processPoint(block, nextBlock, si, n);
24  }
25  if (!nextBlock.isEmpty()) {
26    recurse(nextBlock, n.child1);
27    recurse(nextBlock, n.child2);
28  }
29  nextBlock.copyUp(block);
30 }
```

Fig. 5.2. SIMDized point blocking

`vec_valid`. `vec_valid` is passed to `truncateVec` and `updatePointVec` so that operations are correctly applied according to the original loop's control flow. Finally, any non-truncated points are added to the next block.

Note that strip-mining introduces cleanup code (lines 19–24) that operates over less than a full packet of points. However, as long as there is more than one point, it is still profitable to use SIMD execution. `getValidMask` computes a mask where the bits of valid

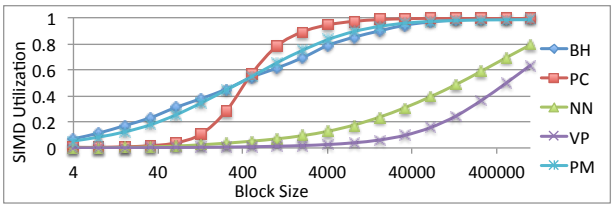


Fig. 5.3. SIMD utilization versus block size ($S = 4$)

points are set. This mask is passed to `processVec` to ensure that only valid points are added to the next block. If the cleanup loop only has one point, we call `processPoint` instead.

The process of replacing the strip-mined inner loop with a SIMD implementation is that of vectorizing a dense loop using techniques such as if-conversion. These transformations can be performed by vectorizing compilers [31].

SIMD utilization

We have referred to the concept of SIMD utilization in prior sections; here we formally define it. Let W be the total amount of work (*i.e.*, the total number of nodes accessed by all the points). Let W_s be the the amount of work done in full SIMD packets (*i.e.*, S times the number of calls to `processVec` in line 17). We can define SIMD utilization $U = W_s/W$[1]. If all points follow the exact same traversal, there will be no divergence and all points can be processed as part of a full SIMD packet; hence, $U = 1$. Figure 5.3 shows the SIMD utilization for five tree traversal algorithms[2]. A larger block size results in better SIMD utilization as there are more points to search through to create full SIMD packets.

Note that a block size equal to the total number of points yields the maximum, *ideal*, SIMD utilization. This schedule of computation extracts the most SIMD potential from the

[1]Note that it is also possible to exploit SIMD in the cleanup loop when it has two or more points; we focus on full SIMD utilization for simplicity.

[2]The **Dragon** input is used for PM, and **Random** inputs are used for the other benchmarks (see Section 6.2). Each input has one million points.

algorithm and input, producing as many full SIMD packets as possible. This ideal utilization measures inherent properties of the algorithm and input. Some algorithms feature so much divergence between traversals that even with extremely large block sizes, it is difficult to find full SIMD packets to process. For example, in NN, the ideal utilization reaches only 0.8. This is because the algorithm itself is highly divergent, and there are many nodes in the tree that fewer than 4 points ever access.

It is clear that in order to maximize SIMD utilization we would like to use a large block size. However a block size that is too large has adverse effects on locality [6] (by way of analogy, think of choosing an overly-large tile size for tiled matrix multiply). Unfortunately, the block sizes at which maximum SIMD utilization is attained feature exactly this poor locality. SIMD cannot help us when the program is memory bound and stalled on load/stores, so poor locality directly counteracts any benefits from vectorization. A schedule that can attain SIMD utilization without compromising locality is necessary.

5.3 Traversal splicing to enhance utilization

In Section 5.1, we described how locality in tree traversals can be cast as a thread scheduling problem, where we would like to schedule threads which access similar tree nodes consecutively for better locality. The same scheduling problem arises in maximizing SIMD utilization. We would like to schedule multiple threads to access the same tree node consecutively so that they can be executed simultaneously with SIMD instructions.

We refer to an example for illustration. Figure 5.4(a) shows an iteration space diagram for two SIMD packets of four threads (*i.e.,* points) each (for the tree in Figure 2.3(a)). Each thread accesses a different set of nodes. For example thread A accesses the entire subtree rooted at ②, but is truncated at ③ (due to truncation condition at line 8 of Figure 2.1), and does not traverse further to ⑥ or ⑦. There are three distinct sets of traversals, with different behaviors: threads $\{A, C, F, H\}$, $\{B, D, G\}$, and $\{E\}$.

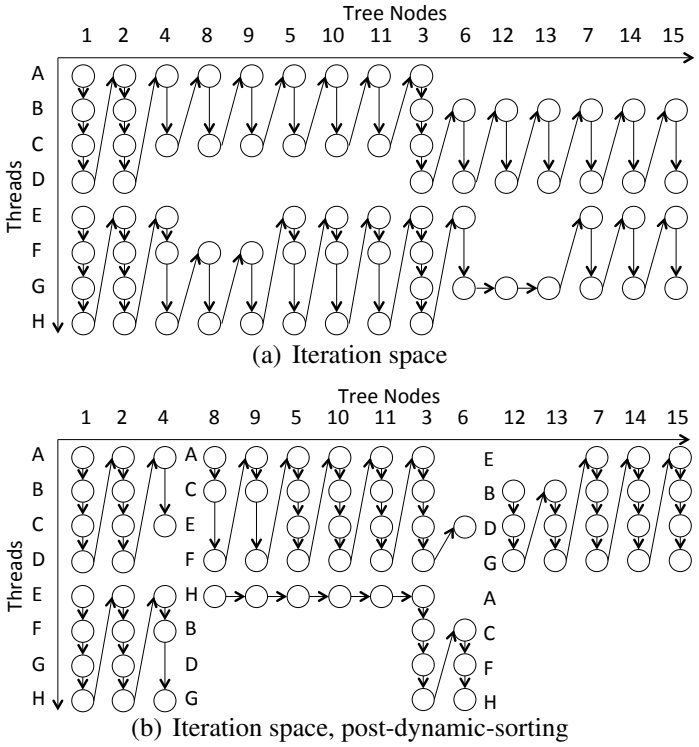


Fig. 5.4. Iteration space of dynamic sorting

SIMD utilization is poor because these distinct traversals are interleaved. At ⑧, both packets have two points. If the threads were scheduled so that A, C, F, H are executed consecutively, ⑧ could be processed with a single *full* packet of four points.

Prior work has used various sorting heuristics to schedule similar traversals together [17, 22]. Unfortunately such *a priori* sorting is highly application specific and can require not only semantic information but substantial programmer effort. We would like an application-agnostic scheduling technique.

5.3.1 Dynamic sorting

As described in Section 5.1.2, traversal splicing enables *dynamic sorting* that uses a thread's past behavior, rather than semantic knowledge, to continuously reorder threads so those with similar traversals will be grouped together. Figure 5.4(b) shows how this

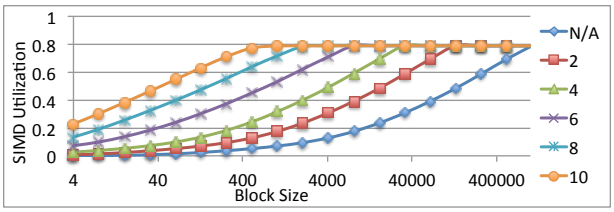


Fig. 5.5. SIMD utilization with dynamic sorting for Nearest Neighbor

dynamic sorting can be applied to the iteration space of Figure 5.4(a). We designate ④ and ⑥ as splice nodes, at which each thread should be paused and reordered. Two SIMD packets traverse up to the first splice node ④ in their original order. At ④, the threads are reordered based on which node each thread last reached. Threads A, C, E, F and H, which reached ④, are moved to the front, and threads B, D and G, which were truncated at ②, are moved to the end. These threads form two new SIMD packets which traverse up to the second splice node ⑥. Then all threads are reordered again, with threads E, B, D and G, which reached ⑥, moved to the front. These threads again form a new SIMD packet.

Dynamic sorting naturally groups threads with similar traversals together. For example at ⑦, whereas previously we had two partial SIMD packets of two points each, we now have a single full SIMD packet of four points, and hence more efficient SIMD execution. This can be generalized beyond packet-based SIMD to point blocking with a block size larger than 4. Dynamic sorting groups similar points together so that more points can be compressed to yield more full SIMD packets. Crucially, *this scheduling is performed without any application-specific knowledge*.

The primary advantage to using dynamic sorting to maximize SIMD utilization is that during the sort phase at each splice node, *all points* can be examined to rearrange the traversals, not just those in a particular block. This yields the same benefits as performing compaction on a very large block of points. However, because splicing only performs this reordering at splice nodes, locality is not harmed. Between splice nodes, smaller block sizes yield good locality.

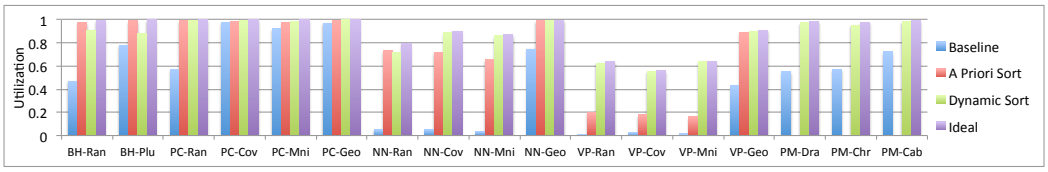


Fig. 5.6. Comparison of SIMD utilization

5.3.2 Improved SIMD utilization

Recall that *ideal* SIMD utilization is defined as the utilization achieved when all threads are available for scheduling at each tree node. Figure 5.5 shows the SIMD utilization for pure point blocking (N/A), and point blocking combined with traversal splicing with splice nodes placed at various uniform depths[3] in the tree for Nearest Neighbor (NN). The dynamic sorting of traversal splicing can significantly improve utilization, so that we need not use as large a block size to get close to ideal. By using traversal splicing, utilization can be improved without sacrificing locality.

Figure 5.6 compares the SIMD utilization of various implementations of our benchmarks (see Section 6.2 for a description of the benchmarks and inputs). We compare the baseline utilization to the utilization of three variants: (i) application-specific, *a priori sorting*[4]; (ii) automatic, *dynamic sorting* using traversal splicing[5]; and (iii) *ideal* utilization. The baseline and variants (i) and (ii) use a block size of 512 (a reasonable size to maintain locality).

It is apparent that dynamic sorting substantially improves SIMD utilization from the baseline, and is competitive with *a priori* sorting. For Vantage Point (VP), dynamic sorting is *substantially better* than *a priori* sorting, illustrating the advantages of traversal splicing. Most importantly, for most benchmarks dynamic sorting nears ideal utilization. In other

[3]While arbitrary splice node placement is possible, we consider only uniform depth placement as it can be implemented more efficiently.

[4]Photon Mapping (PM) is naturally sorted in order of light source, so there is no separate *a priori* configuration

[5]Using empirically determined optimal splice depths as shown in Table 6.1 for Opteron.

words, *dynamic sorting can automatically extract almost the maximum amount of SIMD utilization from our benchmarks*.

5.3.3 Extensions to traversal splicing

In Section 3.2, we found that combining point blocking and traversal splicing was frequently not useful for locality; as a result, point blocking was not applied to all partial traversals in traversal-spliced code. Because our goal is to SIMDize applications, we extend traversal splicing to support applying blocking to all phases.

Some algorithms dynamically generate new points, such as reflected and refracted photons in Photon Mapping. We handle these new points by deferring them to a subsequent outer-loop iteration. Essentially, we treat the root node of the tree as another splice node: as new points visit the root, they are paused until all earlier phases of the computation have finished, after which any points paused at the root begin traversing again. Dynamically generated points tend to have worse coherence than initial points, and dynamic sorting can be even more effective in increasing utilization for such points.

6. AUTOMATIC VECTORIZATION WITH SIMTREE

We implemented our transformations in a source-to-source C++ compiler called SIMTREE[1]. SIMTREE is built on top of the ROSE compiler infrastructure[2].

6.1 Implementation of transformation

SIMTREE applies point blocking and traversal splicing as in the Java transformation framework, TREESPLICER[3], with the extensions described in Section 5.3.3. A key addition in SIMTREE to the features implemented in TREESPLICER is a layout transformation to facilitate SIMDization that changes the layout of the points within a block from an array of structures (AoS) to a structure of arrays (SoA). SoA layout has two advantages: (i) vector load/stores (with SIMD instructions) can be used on packed data; and (ii) packed data has better spatial locality. The disadvantage is that adding points to a point block (*e.g.*, lines 7–10 in Figure 5.2) has higher instruction overhead, as the actual point data must be copied, instead of just a pointer to the point structure.

A whole-program AoS-to-SoA layout transformation is challenging to automate in general, because it is difficult to ensure correctness in the presence of arbitrary aliasing. We overcome this difficulty by limiting the scope of the layout transformation to the traversal code only, copying point data in to SoA-formatted blocks before the traversal and copying from those blocks back to the original layout for the remainder of the code. We need not consider aliasing between points, as the parallel point loop precludes destructive aliasing.

SIMTREE uses an inter-procedural, flow-insensitive analysis to identify which fields must be part of the SoA-formatted block. The necessary fields are those transitively accessed through loop-variant arguments to the recursive call in the enclosing point loop. For

[1] `https://engineering.purdue.edu/plcl/simtree/`
[2] `http://rosecompiler.org`
[3] `https://engineering.purdue.edu/plcl/treesplicer/`

example, in Figure 6.1, the first argument to `recurse` in line 23 is loop variant, and hence a point argument, while the second is not. SIMTREE identifies `f1` and `f2` as point fields (as they are accessed through point argument `p` in lines 11 and 16 respectively), but not `f3`. SIMTREE allocates space in the SoA block for the point fields (Figure 6.2(b)) and transforms the corresponding field accesses in the recursive code to access the data in SoA form.

Because the point data is copied to new storage during the traversal, incorrect values may be computed if there are alternate access paths to access the point data that are not transformed to use the copied data. Rather than performing a complex alias analysis to identify and transform all such access paths, SIMTREE adopts a conservative, field based approach. The copy-in/copy-out approach is safe as long as (i) point fields read during traversal are not written via other access paths and (ii) point fields written during traversal are not read or written via other access paths.

SIMTREE uses the same field analysis to identify any fields accessed via *non-point-based* access paths in the recursive code. In Figure 6.1, `f1` is read via a non-point argument in line 11, while `f3` is read via a non-point argument in line 16. SIMTREE deems the transformation safe as the only field accessed both through points and non-points, `f1`, is read-only.

If there is a conflict between the field accesses, SIMTREE does not apply the SoA transformation. For example, if line 17 in Figure 6.1 were uncommented, then the SoA transformation may not be safe: while the write to `f3` in line 17 through `p` will access the SoA block, the untransformed read of `f3` in line 16 will access the original storage. If there is aliasing between the `point` field of `TreeNode` and any of the `Points` in the array, these reads and writes may be inconsistent. While a more complex alias analysis [52] may more precisely determine the safety of SIMTREE's SoA transformation, this conservative approach is sufficient to prove the transformation safe in all of the applications we have studied.

Figures 6.2(a) and 6.2(b) show concrete code for the block before and after SIMTREE's SoA transformation. In addition to the new arrays for the point fields, the block has two

```
1 struct Point {
2   float f1, f2, f3;
3 }

5 struct TreeNode {
6   TreeNode *child1, *child2;
7   Point *point;
8 }

10 bool truncate(Point *p, TreeNode *n) {
11   if (p->f1 == n->point->f1) return true;
12   return false;
13 }

15 void updatePoint(Point *p, TreeNode *n) {
16   p->f2 += n->point->f3;
17   // p->f3 = n->point->f1;
18 }

20 TreeNode *root;
21 Point* points[N];
22 for (int i = 0; i < N; ++i)
23   recurse(points[i], root);

25 void recurse(Point *p, TreeNode *n) {
26   if (truncate(p, n)) {
27     updatePoint(p, n);
28     return;
29   }
30   recurse(p, n->child1);
31   recurse(p, n->child2);
32 }
```

Fig. 6.1. Concrete example traversal code

`add` functions. The first copies point data to the top level block prior to the traversal (the initial translation from AoS to SoA), and the second adds to the "next block" prior to a recursive call (lines 7–10 of Figure 5.2). This second `add` tracks the index of the source block in a `map` array so that as recursive calls return, writes can be correctly propagated back up the stack by calling `copyUp` (line 29 of Figure 5.2). `copyUp` uses `map` to copy

```
1 struct Block {
2   Point *p[maxSize];
3   int size;
4   void add(Point *point) {
5     p[size++] = point;
6   }
7 }
```

(a) Block as array of structures

```
1 struct Block {
2   Point *p[maxSize]; // allocated for top level only
3   float f1[maxSize];
4   float f2[maxSize];
5   int map[maxSize];
6   int size;
7   void add(Point *point) {
8     p[size] = point;
9     f1[size] = point->f1;
10    f2[size] = point->f2;
11    ++size;
12  }
13  void add(Block *upBlock, int i) {
14    f1[size] = upBlock->f1[i];
15    f2[size] = upBlock->f2[i];
16    map[size] = i;
17    ++size;
18  }
19  void copyUp(Block *upBlock) {
20    for (int i = 0; i < size; ++i) {
21      upBlock->f2[map[i]] = f2[i];
22    }
23  }
24  void copyBack() {
25    for (int i = 0; i < size; ++i) {
26      p->f2 = f2[i];
27    }
28  }
29 }
```

(b) Block as structure of arrays

Fig. 6.2. Block code

written fields (*e.g.,* `f2`) up the stack. `f1` need not be copied back as it is only read. At the top level, `copyBack` (inserted after line 5 of Figure 3.2) copies the block back to the original point storage (translating from SoA back to AoS).

6.2 Evaluation

To demonstrate the efficacy our SIMD transformations, we study five tree traversal algorithms from various domains ranging from scientific applications to data-mining to graphics[4]. The benchmarks are:

1. **Barnes-Hut (BH)** (413 LoC (lines of code)) is a scientific kernel for performing n-body simulation [3]. We use two inputs. *Random* is one million randomly generated bodies. *Plummer* is one million bodies generated from a Plummer model.

2. **Point Correlation (PC)** (285 LoC) is a data mining kernel for finding the number of pairs of points in a dataset that lie within a given radius of each other [38]. We use four inputs. *Random* has $1{,}000{,}000$ randomly generated points in 3 dimensions. *Covtype* is forest cover data, and *Mnist* is handwritten digits data [46], each with $200{,}000$ points reduced by random projections to 7 dimensions. *Geocity* is city coordinates data with $200{,}000$ points in 2 dimensions.

3. **Nearest Neighbor (NN)** (327 LoC) is a data-mining kernel for finding closest points in metric spaces. We use the same inputs as PC, with separate training and test sets of $200{,}000$ points each.

4. **Vantage Point (VP)** (262 LoC) is nearest-neighbor using a vantage-point tree [53] rather than a kd-tree. We use the same inputs as NN.

5. **Photon Mapping (PM)** (8498 LoC) is described in detail in Section 2. We use three inputs: *Dragon* has $100{,}000$ triangles, *Christmas* has $1{,}091{,}067$ triangles and *Cabin*

[4]We do not use exactly the same benchmarks as in our prior work [6, 7], as SIMTREE targets C++ rather than Java.

has $422{,}735$ triangles. Each scene was rendered with 10 lights, and $100{,}000$ photons per light.

BH, PC and NN are adapted from the Lonestar suite [45], and VP[5] and PM[6] are adapted from open source implementations. We evaluate seven variants of each benchmark:

1. **Base**: the baseline described for each benchmark above.
2. **Block4**: point blocking with a block size of 4.
3. **Block4+SIMD**: packet based SIMD traversals [5, 32]. This is equivalent to SoA and SIMD applied to **Block4**.
4. **Block**: point blocking with an autotuned block size [6][7].
5. **Block+SIMD**: SoA and SIMD applied to **Block**.
6. **Block+Splice**: traversal splicing with dynamic sorting applied to **Block** [7]. We place splice nodes at an empirically determined optimal splice depth for each application/input/platform combination as given in Table 6.1.
7. **Block+SIMD+Splice**: SoA and SIMD applied to **Block+Splice**.

Our baseline benchmarks are true baselines: no a priori *sorting is performed.* SIMDization is performed manually, using standard techniques such as if-conversion. While the SIMDization can be done automatically by a vectorizing compiler, such compilers may miss readily-exploitable SIMD opportunities [31], as discussed in Section 6.2.5.

The benchmarks were written in C++ and compiled with gcc 4.4.6 with -O3. Each configuration was run 4 times enforcing a coefficient of variation[8] less than 0.02, by extending the number of runs until steady state if necessary. This yields errors of at most $\pm 3.18\%$ of the mean with 95% confidence.

[5] `http://stevehanov.ca/blog/index.php?id=130`

[6] `http://code.google.com/p/heatray`

[7] We use 1% of the points to test various block sizes 5 times each, and choose the best block size. This results in a block size of 128 for the **Covtype**, **Mnist** and **Geocity** inputs, and a block size of 512 for all other inputs.

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

Benchmark	Input	Average Reach	Splice Depth		
			Xeon	Opteron	Auto-0.5
Barnes-Hut	Random	5.88	2	2	3
	Plummer	6.88	3	3	3
Point Correaltion	Random	17.78	8	8	9
	Covtype	15.86	7	8	8
	Mnist	15.23	7	7	8
	Geocity	15.94	9	8	8
Nearest Neighbor	Random	14.82	9	9	7
	Covtype	14.95	9	9	7
	Mnist	14.88	9	9	7
	Geocity	15.86	8	7	8
Vantage Point	Random	19.62	9	10	10
	Covtype	17.39	9	9	9
	Mnist	17.40	9	9	9
	Geocity	17.19	8	8	9
Photon Mapping	Dragon	9.13	7	8	5
	Christmas	12.55	15	13	6
	Cabin	13.14	17	18	7

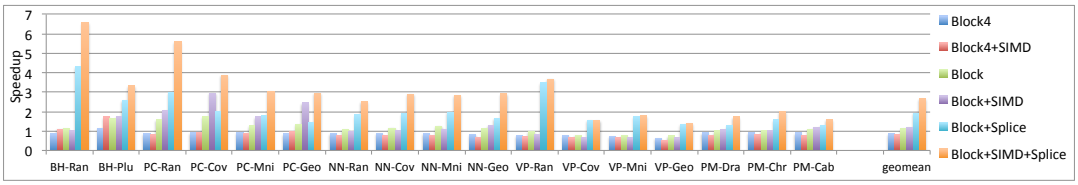
Table 6.1
Optimal and autotuned splice depths

We used two platforms for evaluation:

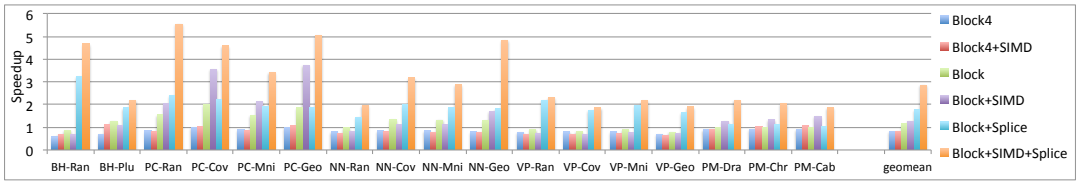
- **Xeon** runs Linux 2.6.32 and contains four eight-core Intel Xeon E5-4650 chips.
- **Opteron** runs Linux 2.6.32 and contains four sixteen-core AMD Opteron 6282 chips.

6.2.1 Speedups compared to baseline

We will first examine the *single-thread* speedups of our transformed versions compared to the baseline, shown in Figure 6.3. The rightmost bars are the geometric means of all 17 benchmark/input sets. We also measured multi-thread runs where both SIMD and threads are utilized, and found that SIMD improvements scale well with multiple threads: SIMD augments the parallelism achievable from multithreaded execution. We do not show these results due to space limitations.



(a) Xeon

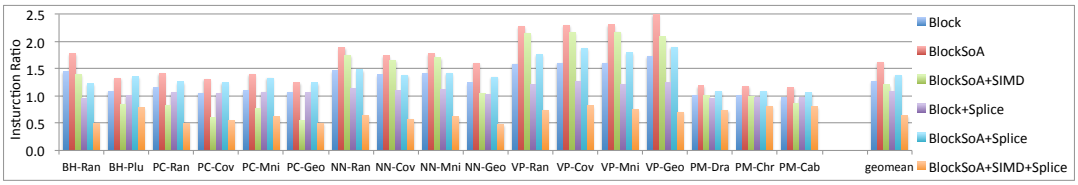


(b) Opteron

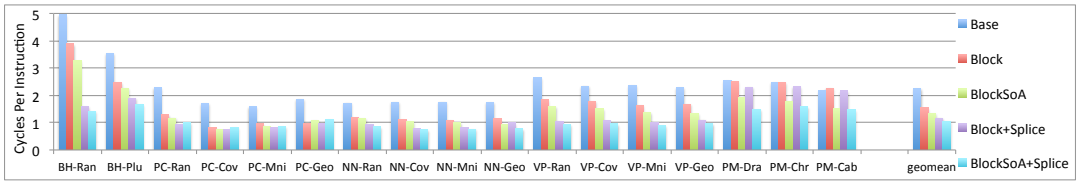
Fig. 6.3. Speedup over **Base**

Block4 and **Block4+SIMD** perform similarly or worse than the baseline. This is because a block size of 4 does not improve locality and only adds additional overhead for point blocking. Applying SIMD fails to provide improvements because the naïve packet based SIMD implementation has poor utilization as illustrated in Figure 5.3. **Block** is able to provide small speedups from enhanced locality—on average, 1.127 and 1.154 on Xeon and Opteron, respectively. **Block+SIMD** provides additional gains for speedups of 1.188 and 1.266 respectively. This is because point blocking with compression improves utilization so that SIMD execution is profitable.

Block+Splice adds the locality benefits of traversal splicing to **Block** and results in larger speedups of 1.920 and 1.783. **Block+SIMD+Splice** adds SIMD to **Block+Splice** for substantially larger speedups of 2.689 and 2.864. The average improvement from adding SIMD to **Block+Splice** is much larger than from adding SIMD to **Block**, because the dynamic sorting of traversal splicing dramatically improves utilization, as illustrated in Figure 5.6. For example in NN-Random, adding SIMD to **Block** *degrades* performance because SIMD execution is unable to make up the instruction overhead of the structure-of-arrays layout transformation. This is unsurprising as SIMD utilization is only 0.052 even



(a) Instruction counts normalized to **Base**



(b) Cycles per instruction

Fig. 6.4. Performance counters

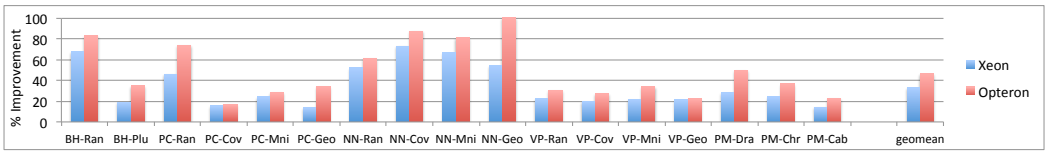


Fig. 6.5. Improvement in SIMD performance from dynamic sorting

for a block size of 512. However dynamic sorting improves utilization to 0.716, making SIMDization of **Block+Splice** profitable.

6.2.2 Performance counters

Our optimizations achieve speedups because the locality and SIMD execution benefits they provide outweigh the costs incurred by our scheduling and layout transformations. To analyze this cost-benefit tradeoff in more detail, we collected performance counter results on Opteron with PAPI[9].

[9] `http://icl.cs.utk.edu/papi/`

One indicator of the effectiveness of SIMDization is instruction count reduction. Figure 6.4(a) shows the instruction counts normalized to the baseline. **BlockSOA** denotes the structure of arrays (SoA) transformation added to **Block**. On average, point blocking results in instruction overhead of 1.266. The SoA transformation increases this overhead to 1.617. Adding SIMD decreases instruction overhead to 1.204 (as SIMD replaces multiple instructions with a single instruction). Adding traversal splicing to point blocking *reduces* instruction overhead to 1.079 (from 1.266), because the dynamic sorting makes the blocks denser and results in fewer blocks traversing the tree. Finally adding SoA increases the instructions to 1.380, but then adding SIMD halves it to 0.643.

The savings in instruction overhead from adding SIMD is much larger with traversal splicing ($2.14\times$) than without ($1.34\times$), as traversal splicing significantly enhances SIMD utilization. We do not expect instruction counts to decrease by a full-SIMD-width factor of 4 because: (i) we measure instruction counts for the whole traversal, while SIMD only affects the computation portion; (ii) SoA adds instruction overhead; and (iii) SIMD utilization is less than 1.

Figure 6.4(b) shows the cycles per instruction (CPI) of the transformations with and without SoA layout and/or traversal splicing. CPI is a proxy for locality: a program with poor locality will have higher CPI[10]. The average baseline CPI is 2.243. Point blocking and traversal splicing reduce this to 1.563 and 1.170 respectively through enhanced locality. The SoA layout also improves spatial locality as the point data is packed into the blocks instead of being spread out over the heap. This further reduces CPI to 1.349 and 1.043 when applied on top of point blocking and traversal splicing respectively. Hence, a combination of instruction decrease through SIMD execution, and enhanced locality through SoA layout accounts for our performance improvements.

[10]This CPI comparison does not use SIMD; we expect **BlockSOA** and **BlockSOA+SIMD** to have similar locality.

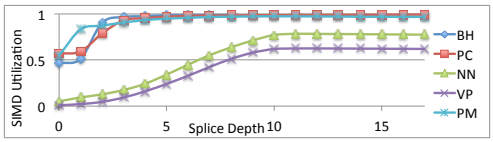


Fig. 6.6. SIMD utilization for various splice depths ($B = 512$)

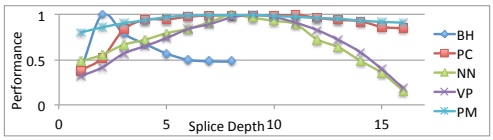
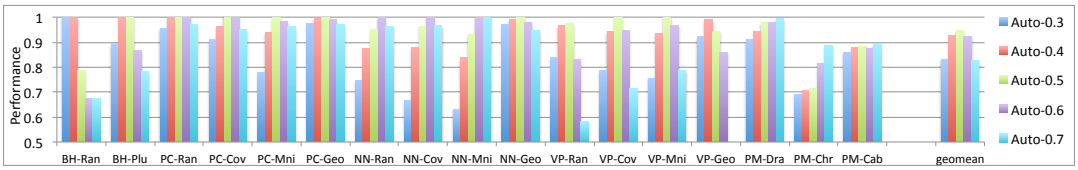


Fig. 6.7. Performance normalized to optimal splice depth ($B = 512$)

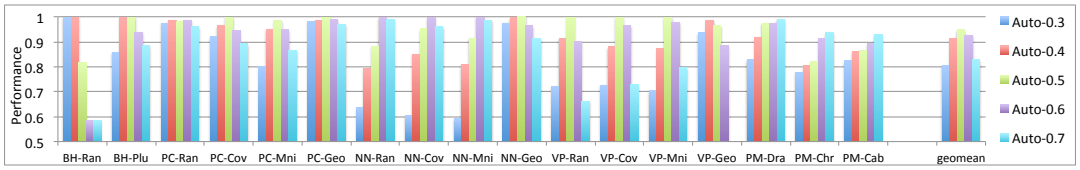
6.2.3 SIMD performance improvement from dynamic sorting

Because traversal splicing provides both locality benefits and SIMD utilization benefits, we isolate how much dynamic sorting in particular improved SIMD performance. Let us define the speedups of **Block**, **Block+SIMD**, **Block+Splice** and **Block+SIMD+Splice** compared to the baseline as B_b, B_{bs}, B_{bp} and B_{bsp} respectively (where p is used for S*p*lice). The benefit of applying SIMD to **Block** and **Block+Splice** can be defined as $S_b = B_{bs}/B_b$ and $S_{bp} = B_{bsp}/B_{bp}$, respectively. The ratio of SIMD performance improvements in both cases, $D = S_{bp}/S_b$, lets us quantify how much dynamic sorting improves SIMD quality independent of locality effects (which apply to both SIMD and non-SIMD variants).

Figure 6.5 shows D in % improvement for the two systems. Dynamic sorting is ineffective for BH-Plummer because the baseline has a high utilization of 0.779, and dynamic sorting is able to improve that only to 0.882 (Figure 5.6). Dynamic sorting is very effective for NN-Geo because the baseline has a utilization of 0.747 which dynamic sorting improves to 0.995, nearly perfect utilization. On average D in % improvement is 33.0% and 46.4% respectively for Xeon and Opteron.



(a) Xeon



(b) Opteron

Fig. 6.8. Performance at automatically selected splice depth normalized to performance at optimal splice depth

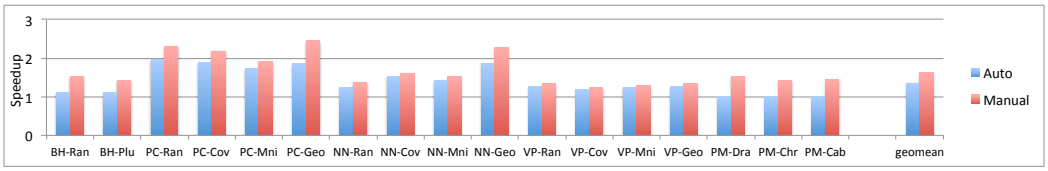


Fig. 6.9. SIMD speedup from automatic vectorization

6.2.4 Automatically selecting splice depth

The results discussed thus far have used an autotuned block size with an empirically determined optimal splice depth. We would like to automatically select the splice depth as well, to fully automate our transformation.

While a good block size can be autotuned by using a small portion (1%) of the points to test blocks of varying sizes, and choosing the best performing block size [6], a similar approach is infeasible for traversal splicing because we would like to apply splicing over the full set of points.

When considering locality, there is a tradeoff in splice depth placement: placing splice nodes too shallow or too deep results in worse locality, and there is a sweet spot in the

middle [7]. On the other hand, utilization increases almost monotonically with splice depth, as deeper and more numerous splice nodes allow for more dynamic sorting[11]. Figure 6.6 shows utilization for varying splice depths, with a fixed block size of 512 (using the same inputs as in Figure 5.3). We expect to see the best performance when striking the right balance between locality and SIMD utilization.

Figure 6.7 shows the performance at varying splice depths normalized to the performance obtained with the optimal splice depth for each benchmark. We see a clear peak where utilization and locality strike a good balance, surrounded by splice depths which perform worse.

In prior work we demonstrated that combining point blocking with traversal splicing reduces a program's sensitivity to splice depth, and found empirically that a depth at half of the average reach (the depth of the nodes where a point's recursion is stopped) is a good splice depth [7]. We evaluate a range of parameters for splice depth, ranging from 0.3 to 0.7 times the average reach. Figure 6.8 shows the performance of these parameters compared to the performance at an optimal splice depth. Table 6.1 shows the average reach of each benchmark/input, the optimal depth on each system, and the automatically selected depth at half of the average reach. It can be seen that **Auto-0.5** performs best, attaining 95% of optimal performance on average on both systems.

6.2.5 Automatic vectorization with icc

SIMTREE does not generate SIMD code. As described previously, SIMTREE's goal is to generate *vectorizable* code; the resulting inner loop (line 17 of Figure 5.2) must be vectorized in a separate pass. To extract maximum SIMD performance in our results, we vectorized this loop by hand. We also investigated the ability of icc (the Intel compiler) to do the vectorization. Because the presence of any non-vectorizable code or nested loops precludes vectorization by icc [31], we applied a combination of loop distribution and interchange to split the target loop into a series of icc-vectorizable loops (this process can

[11] Utilization can decrease if the splice depth is deeper than the tree itself.

be readily automated as the target loop is fully parallel). Figure 6.9 shows the performance of icc-vectorized code and hand-vectorized code for the main SIMD loop (ignoring the cleanup loop), relative to the SIMTREE-transformed baseline (which applies blocking, splicing and SoA) on Xeon. We see that auto-vectorization attains a 1.354 speedup, while our hand vectorization attains a 1.618 speedup; auto-vectorization attains on average 84% of the hand-vectorized performance.

7. RELATED WORK

7.1 Locality in non-tree irregular algorithms

Much of the work on optimizing locality in irregular algorithms has focused on *scheduling* computation so that tasks likely to access similar data are scheduled in close succession to exploit temporal locality. This has been the strategy of choice for optimizing sparse-matrix algorithms [25–27], where most approaches use an *inspector-executor* approach to scheduling. The structure of the computational tasks is found in an inspection phase, which rearranges them to improve locality. The rearranged schedule is then executed. Inspector-executor approaches are less useful for tree traversal codes, as the inspection phase requires performing the traversals, incurring all the misses we hope to avoid.

7.2 Point sorting

Scheduling approaches for tree traversals (the "sorting" optimizations we discuss in Section 2.2) have instead used semantic knowledge to schedule the points without performing the traversals, often with space filling curves [16, 22]. Salmon used Orthogonal Recursive Bisection [54] to directly partition the point space to provide physical locality [55]. Singh *et al.* recognized that N-body problems already have a representation of the spatial distribution encoded in the tree data structure and partitioned the tree instead of partitioning the point space directly [17]. Mansson *et al.* propose various sorting optimizations for reflected rays in ray tracing [19]. In fact sorting for ray tracing is difficult and important enough that Moon *et al.* first construct and traverse a *simplified* scene, and sort rays based on hit points of the simplified scene [22].

7.3 Application specific work similar to point blocking

There have been a few application-specific, manual approaches similar to point blocking and traversal splicing. Point blocking-like approaches arise in vectorization where a block, or vector, of points are processed simultaneously on vector hardware. Hernquist vectorized Barnes-Hut across nodes of the tree, so that each point traverses all nodes at the same level simultaneously [56]. This approach effectively changes the order of the tree traversal from depth-first to breadth-first. This has two drawbacks. First, it changes the traversal order of the tree, affecting the result in the presence of non-commutative operations (such as floating-point addition). Second, there typically are not many nodes per tree level, leading to short vectors (and less parallelism). Makino vectorized the tree traversal across points, instead, leading to a per-point parallelization similar to the loop interchanged implementation described in Section 2.1 [57]. However, there are a few key points to note. First a simple loop interchange does not suffice to exploit locality. Second, Makino's transformation relies on a pre-computed traversal of the tree, and changes the order in which particular tree nodes are visited by different points, reducing the generality of his transformation. Wald *et. al.* presented an interactive ray tracer by constructing packets (*i.e.,* blocks) of four rays (*i.e.,* points), and using SIMD traversals on packets instead of individual rays [5]. Their approach is equivalent to point blocking with a block size of 4.

7.4 Application specific work similar to traversal splicing

Application-specific, manual approaches similar to traversal splicing, have mostly targeted locality for ray tracing. Pharr *et al.* partition the scene into a uniform grid of "voxels" [20]. As rays traverse a scene, they pause at the boundaries of voxels and are resumed later. By processing rays on a per-voxel basis, locality is improved. They schedule voxels based on a cost-benefit heuristic which considers the amount of geometry already cached and the number of the rays paused at the voxel. Navrátil *et al.* pause traversals at "queue points" (akin to splice nodes) in the tree [21]. Queue points are placed so that its subtree fits entirely in L2 cache. Navrátil's thesis focuses on dynamic scheduling of rays to enhance lo-

cality for memory efficiency and scalability across thousands of distributed processors [1]. Aila and Karras extend Navrátil *et al.*'s work to hierarchical queue points and massive parallelism [23]. They use dynamic programming to place treelets (subtrees at which rays are paused) with balanced surface area, hence minimizing treelet transitions per ray, and discuss considerations for a dedicated GPU architecture which supports efficient pausing and resuming of rays. These approaches are able to make smarter decisions on splice node placement and point scheduling (*e.g.,* next voxel to process) with semantic knowledge of the algorithm. We believe TREESPLICER could potentially make such smart decisions automatically without semantic knowledge, through a dynamic analysis of traversal patterns at runtime.

Outside the graphics domain, Ghoting *et al.* propose *path tiling* for frequent pattern matching, where the tree is partitioned into tiles, and computation is paused at tile boundaries (*i.e.,* splice nodes) to enhance temporal locality [24]. Pingali *et al.* propose *computation reordering*, whereby an individual computation can be paused during its execution and coalesced with other computations that are accessing the same part of the data structure [18]. However, computation reordering is more a set of principles for optimization than an optimization itself: correctly applying reordering requires manually transforming algorithms in application-specific ways. Point blocking and traversal splicing can be seen as a special, disciplined case of computation reordering, applying to tree traversals, that can be implemented automatically and efficiently.

7.5 Analyses for tree codes

Aluru *et al.* discussed changing the tree structure of Barnes-Hut to improve performance [58]. We note that our transformations are independent of the type of tree used (indeed, the tree in raytracing is different from that in Barnes-Hut), and hence our approach can apply to their algorithm as well. Rinard and Diniz used a commutativity analysis to parallelize an N-body code in a unique manner [59]. Rather than distributing the points among threads, they are able to prove through compiler analysis that updates to the points

commute, and hence multiple threads can update points simultaneously. This is akin to parallelizing the traversal loop in our abstract model, rather than the point loop. Ghiya *et al.* proposed an algorithm to detect parallelism in C programs with recursive data structures [60]. These tests rely on shape analysis to provide information on whether the data structure is a tree, DAG or general graph, and apply different dependence tests depending on data structure shape. Their analyses focus on parallelization and do not consider locality, but we believe their approaches might inform an automatic transformation framework that implements our techniques.

7.6 Spatial locality in irregular algorithms

A large number of prior studies have investigated improving *spatial* locality in irregular algorithms. Truong *et al.* propose *ialloc* to facilitate programmer driven field reorganization and instance interleaving, where frequently used "hot" fields of many instances are interleaved, so that "cold" fields do not occupy the same cache line as hot fields [11]. Chillimbi *et al.* automate this process for Java programs with a profiling and program transformation tool, and automatically generate field reordering recommendations for C programs [13]. Separate work by Chillimbi *et al.* propose *ccmorph* to reorganize memory layout of trees and *ccmalloc* to collocate objects based on programmer provided hints [12], and perform relayout during garbage collection [10]. Lattner and Adve use a context-sensitive pointer analysis to segregate distinct instances of heap allocations into separate memory pools, and thereby enhance locality at the *macroscopic* level of entire data structures [14]. Wang *et al.* also aggregate affinitive heap objects into dedicated memory regions, but do so dynamically without access to the source code [15]. Because these approaches focus on data layout, they do not target temporal locality, as the transformations presented in this paper do. We expect our transformations to be complementary to these spatial-locality-enhancing approaches.

7.7 Vectorization of tree traversals

Much of the work on using SIMD for tree traversals has come from the graphics domain. Prior work proposes various techniques for manual, application specific SIMDization of ray-object intersection test traversals. Wald *et. al.* present an interactive ray tracer by using packet-based SIMD traversals on packets of four rays [5]. Such packet-based SIMD traversals work well for coherent eye rays, but break down for secondary rays, which are necessary to model realistic lighting effects. To circumvent this incoherency issue, subsequent work has looked to other sources of SIMD. Dammertz *et. al.* use a quad-BVH (bounding volume hierarchy with four children per node) and SIMDize computation across the four children for each *individual* ray [34]. Pixar used SIMD to test all three dimensions (x, y, z) of a bounding box at once in rendering the movie "Cars" [35]. Havel and Herout propose a modified ray-object intersection algorithm which is better suited for SSE4 which supports a SIMD dot product instruction [36]. They note that applying SIMD to individual rays get only 10-25% improvement whereas applying SIMD to coherent packets can get 300%. Our dynamic sorting can make incoherent packets coherent, and opens up opportunities for substantial performance gains in the ray tracing domain. Furthermore our technique can be applied automatically.

In the database domain, Kim *et. al.* propose FAST (Fast Architecture Sensitive Tree) to accelerate index searches by using SIMD to compare a search value to N keys at once, where N is the number of keys that can fit in a SIMD register [32]. For example, with $N = 4$, a point can compare with a node, the node's left child, and the node's right child at once, and decide which grandchild to move on to. Yamamuro *et. al.* extend FAST by compressing keys at lower depths so that more keys can be compared at once [33]. These techniques capitalize on specific properties of index search: that each point takes a single path from root to leaf, and that the point-key comparison can be done at multiple levels simultaneously. For point correlation, Chhugani *et. al.* propose a novel SIMD friendly histogram update algorithm, exploiting an alternate source of SIMD instead of vectorizing the traversals themselves [38].

Ren *et. al.* focus on applications where points traverse *many* pointer based data structures (*e.g.,* random forests, regular expressions), and *manually* SIMDize a single point visiting nodes of many structures [37]. Kim and Han propose techniques for efficient SIMD code generation for irregular loops with array indirection [61]. Barthe *et. al. synthesize* SIMD code given pre/post conditions for irregular loop kernels in C++ STL or C# BCL [62]. The latter two works do not consider pointer-based irregular structures.

Nuzman and Zaks examine a direct unroll-and-jam approach for vectorization of outer loops which could help auto-vectorize our transformed point loop [63]. Maleki *et. al.* perform a thorough study of modern vectorizing compilers [31]. They find that current compilers are still lacking: xlc, icc and gcc are able to individually vectorize only 18-30% of loops extracted from real codes, while they can collectively vectorize 48%. In particular, they note that traditional vectorizing compilers primarily target regular loops; our transformations hence open traversal codes to automatic vectorization.

8. CONCLUSIONS

This dissertation is the first to develop general compiler transformations and optimizations to automatically enhance the performance of tree traversal algorithms. We investigated two sources of performance improvement: locality enhancement and vectorization. We have developed techniques for both sources, and demonstrated that these techniques can be automatically applied by an optimizing compiler, delivering substantial performance improvements, hence relieving programmers of manual, error-prone, application-specific effort.

8.1 Enhancing locality

We presented point blocking and traversal splicing, two comprehensive, general, automatic transformations that applies to tree traversal codes such as Barnes-Hut and nearest neighbor. These transformations are analogs of the classical loop tiling transformations for tree traversal algorithms: point blocking is equivalent to tiling the point loop so that a point block stays in cache, and traversal splicing is equivalent to tiling the traversal loop so that a partial tree stays in cache.

Point blocking works well when combined with a manual, application specific sorting optimization. Traversal splicing exploits the insight that during a point's traversal of a tree, its remaining traversal structure can be predicted from its past behavior, and performs a semantics oblivious dynamic sorting on the fly. Hence the effectiveness of traversal splicing is not dependent on first performing any application-specific, semantics-aware hand transformation. Often the best performance is attained by combining both point blocking and traversal splicing.

We presented TREESPLICER, a compiler framework which can automatically apply point blocking and/or traversal splicing. TREESPLICER uses autotuning techniques to se-

lect suitable parameters (*e.g.,* block size, splice depth) for our transformations. We demonstrated that for six benchmark algorithms, a combination of point blocking and traversal splicing can deliver single-thread speedups of up to 8.71 (geometric mean: 2.48) over baseline implementations, just from better locality.

8.2 Vectorization

We presented automatic transformations to reschedule tree traversal algorithms to effectively SIMDize them. Our scheduling techniques build on aforementioned techniques which enhanced locality for these algorithms. Point blocking exposes a loop structure within the traversal function which can be SIMDized (as in manually transformed packet-based SIMD traversals), and compacts points to improve SIMD utilization. Traversal splicing pauses points at predesignated nodes, and reorders the points based on their past traversal history. This dynamic sorting groups points with similar traversals together so that SIMD utilization is further enhanced.

We demonstrated that dynamic sorting dramatically improves SIMD utilization to the point that it is very close to ideal. This enhanced utilization results in significantly better SIMD performance, yielding speedups of up to 6.59 (geometric mean: 2.78). Our techniques can deliver SIMD speedups even in algorithms where the incoherency of points have previously forced researchers to look for other sources of SIMD.

LIST OF REFERENCES

LIST OF REFERENCES

[1] P. A. Navratil, *Memory-efficient, scalable ray tracing*. PhD thesis, Austin, TX, USA, 2010.

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

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

[4] A. G. Gray and A. W. Moore, "N-Body Problems in Statistical Learning," in *Advances in Neural Information Processing Systems (NIPS) 13 (Dec 2000)* (T. K. Leen, T. G. Dietterich, and V. Tresp, eds.), 2001.

[5] I. Wald, P. Slusallek, C. Benthin, and M. Wagner, "Interactive Rendering with Coherent Ray Tracing," *Computer Graphics Forum (Proceedings of EUROGRAPHICS)*, vol. 20, no. 3, pp. 153–164, 2001.

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

[7] Y. Jo and M. Kulkarni, "Automatically enhancing locality for tree traversals with traversal splicing," in *Proceedings of the 2012 ACM international conference on Object oriented programming systems languages and applications*, 2012.

[8] Y. Jo, M. Goldfarb, and M. Kulkarni, "Automatic vectorization of tree traversals," in *Proceedings of the 2013 International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, 2013.

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

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

[11] 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*, pp. 322–, 1998.

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

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

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

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

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

[17] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy, "Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity," *J. Parallel Distrib. Comput.*, vol. 27, no. 2, pp. 118–141, 1995.

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

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

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

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

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

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

[24] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y. kuang Chen, and P. Dubey, "Cache-conscious frequent pattern mining on a modern processor," in *In Proceedings of the International Conference on Very Large Data Bases (VLDB*, pp. 577–588, 2005.

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

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

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

[28] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth, "A scalable auto-tuning framework for compiler optimization," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pp. 1–12, 2009.

[29] R. Vuduc, J. W. Demmel, and K. A. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," *Journal of Physics: Conference Xseries*, vol. 16, no. 1, 2005.

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

[31] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua, "An evaluation of vectorizing compilers," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pp. 372–382, 2011.

[32] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "Fast: fast architecture sensitive tree search on modern cpus and gpus," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pp. 339–350, 2010.

[33] T. Yamamuro, M. Onizuka, T. Hitaka, and M. Yamamuro, "Vast-tree: a vector-advanced and compressed structure for massive data tree traversal," in *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pp. 396–407, 2012.

[34] H. Dammertz, J. Hanika, and A. Keller, "Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays," in *Proceedings of the Nineteenth Eurographics conference on Rendering*, EGSR'08, pp. 1225–1233, 2008.

[35] P. Christensen, J. Fong, D. Laur, and D. Batali, "Ray tracing for the movie 'cars'," in *Interactive Ray Tracing 2006, IEEE Symposium on*, pp. 1 –6, 2006.

[36] J. Havel and A. Herout, "Yet faster ray-triangle intersection (using sse4)," *Visualization and Computer Graphics, IEEE Transactions on*, 2010.

[37] B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte, "Simd parallelization of applications that traverse irregular data structures," in *Proceedings of the Eleventh International Symposium on Code Generation and Optimization*, CGO '13, (New York, NY, USA), pp. 1–12, ACM, 2013.

[38] J. Chhugani, C. Kim, H. Shukla, J. Park, P. Dubey, J. Shalf, and H. D. Simon, "Billion-particle simd-friendly two-point correlation on large-scale hpc cluster systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 1:1–1:11, IEEE Computer Society Press, 2012.

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

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

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

[42] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[43] 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*, pp. 1–18, 2007.

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

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

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

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

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

[49] S. M. Omohundro, "Five balltree construction algorithms," tech. rep., 1989.

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

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

[52] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pp. 105–118, 1999.

[53] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, SODA '93, pp. 311–321, 1993.

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

[55] J. K. Salmon, *Parallel hierarchical N-body methods*. PhD thesis, 1991.

[56] L. Hernquist, "Vectorization of tree traversals," *J. Comput. Phys.*, vol. 87, pp. 137–147, March 1990.

[57] J. Makino, "Vectorization of a treecode," *J. Comput. Phys.*, vol. 87, pp. 148–160, March 1990.

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

[59] M. Rinard and P. C. Diniz, "Commutativity analysis: a new analysis technique for parallelizing compilers," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 6, pp. 942–991, 1997.

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

[61] S. Kim and H. Han, "Efficient simd code generation for irregular kernels," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pp. 55–64, 2012.

[62] G. Barthe, J. M. Crespo, S. Gulwani, C. Kunz, and M. Marron, "From relational verification to simd loop synthesis," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pp. 123–134, 2013.

[63] D. Nuzman and A. Zaks, "Outer-loop vectorization: revisited for short simd architectures," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pp. 2–11, 2008.

VITA

VITA

Youngjoon Jo was born in Seoul, Korea in 1983. He came to the U.S. with his family in 1989 and attended first to seventh grade in New Jersey and Minnesota. Then he returned to Seoul and went on to study Electrical Engineering at Seoul National University (SNU). He sang second tenor in the SNU Choir and made many lifelong friends. He received his B.S.E.E. from SNU in 2009. He is a Ph.D. candidate in the School of Electrical and Computer Engineering at Purdue University, and his fabulous advisor is Milind Kulkarni. His research interests are in compilers, systems and mobile. His website is http://youngjoon.net.