Tree-Traversal Orientation Analysis

Kevin Andrusky, Stephen Curial and José Nelson Amaral

Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
{andrusky, curial, amaral}@cs.ualberta.ca

Abstract. This paper presents a profiling-based analysis to determine the traversal orientation of link-based tree data structures. Given the very-high memory-hierarchy latencies in modern computers, once the compiler has identified that a pointer-based data structure represents a tree, it would be useful to determine the predominant orientation of traversal for the tree. Optimizing compilers can implement the static shape analysis proposed by Ghiya and Hendren to determine if a linked data structure is a tree [10]. However no techniques have been reported to enable an optimizing compiler to determine the predominant traversal orientation of a tree. This paper describes an analysis that collects data during an instrumented run to determine if the traversal is predominantly breadth-first or depth-first. The analysis determined, with high accuracy, the predominant orientation of traversal of trees in programs written by us as well as in the Olden benchmark suite. This profile-based analysis is storage efficient — it uses only 7% additional memory in comparison with the non-instrumented version of the code. Determining the predominant orientation of traversal of a tree data structure will enable several client optimizations such as improved software-based prefetching, data-storage remapping and better memory allocators.

1 Introduction

Data locality is critical for performance in modern computers. A fast processor's time is wasted when programs with poor data locality spend a significant amount of time waiting for data to be fetched from memory [1, 12]. Although optimal cache-conscious data placement is NP-hard and difficult to approximate well [17], researchers have developed many techniques that reduce memory stalls in pointer-chasing programs [2, 6–9, 14]. Some improvements to data locality require changes to the source code [7, 11, 18, 16]. An alternative is for an optimizing compiler to automatically perform data transformations that improve locality [2, 9, 14, 20].

Most published pointer-chasing optimizations do not specifically address the problem of improving data locality when the pointer-based data structure represents a tree. This paper presents a profile-based tree-traversal orientation analysis that determines the primary orientation of traversal for each pointer-based

tree in a program. The results of this analysis can be used by several client optimizations to improve locality beyond what is possible with current techniques. This analysis neither requires changes to the source code nor requires significant alterations to existing compilers.

Often, dynamically allocated structures are not allocated in the same order in which they are referenced. For instance the allocation order may be determined by the organization of a data file while the traversal order is determined by the algorithm that uses the data. Data structures may exhibit poor locality due to the structure's design or due to an initial implementation that does not consider locality [11, 16]. As a result, the nodes of a linked data structure may be scattered throughout the heap. For example, assume an application containing a binary tree with 15 nodes, allocated into consecutive memory locations with a breadth-first orientation as shown in Figure 1.
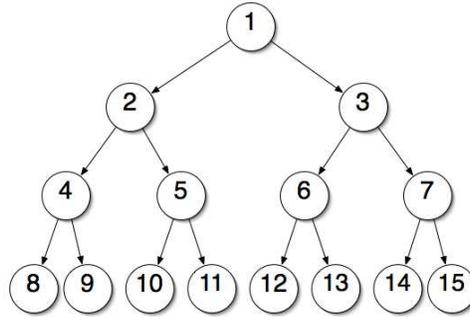

Fig. 1. A binary tree with 15 nodes. The node number indicates the order in which the node was allocated.

If the application accesses these nodes in a depth-first orientation[1] it will exhibit relatively poor spatial locality and likely result in degraded performance. On the other hand, if the data had been allocated in a depth-first fashion, then the traversal would have been a series of accesses to adjacent locations in memory. A regular access pattern would enable latency-hiding techniques, such as prefetching, to greatly reduce the number of memory stalls.

Optimizing compilers are better able to transform programs to improve data locality when enough information about the data access pattern is available. For instance, if a compiler could determine the traversal order of the tree in the example above, it could automatically replace a standard memory allocator with a custom memory allocator that would place tree nodes with strong reference affinity close to each other in memory.

---

[1] Two possible depth-first access sequences are ⟨1, 2, 4, 8, 9, 5, 10, 11, 3, 6 ,12, 13, 7, 14, 15⟩ and ⟨1, 3, 7, 15, 14, 6, 13, 12, 2, 5, 11, 10, 4, 9, 8⟩

There are many ways of traversing and accessing data stored in trees. Most of these fall into two general categories: *breadth-first* and *depth-first* traversals. If the predominant traversal orientation of a tree-based data structure is known, the compiler can reduce memory stalls caused by poor data locality. Sections 2 and 3 of this paper present a method to determine the types of traversals taking place in a program, while minimizing the amount of memory used to make this determination. This profile-based analysis compiles an instrumented version of an application, and executes the profiled code. After the program execution is finished, a value in the range $[-1, 1]$ is returned for each tree-based structure, which indicates the primary orientation of the tree traversal in the profiled code.

In order to determine the effectiveness of this analysis, we wrote several programs containing tree-based data structures with known traversal orientations. We also ran the analysis for several of the programs that comprise the Olden benchmark suite. The results of these tests are presented in Section 4. For all the tests performed, the analysis successfully identified the predominant orientation of traversal of the tree-based data structured. In addition, the amount of memory used by the profiled code, and the runtime of the code, were evaluated. In general, the instrumentation required only 7% more memory on top of the program's normal memory requirements. The instrumented application require approximately six times longer to execute than the non-instrumented version of the program. Given that training runs are typically executed once and offline, the longer execution time is acceptable in most domains.

A large number of client optimizations, in existing compilers or proposed in recent literature, could make use of the information provided by our analysis. In addition to the custom-memory allocator mentioned above, this analysis would be useful with many prefetching and structure-field reordering techniques. Section 5 surveys techniques that are possible clients to our analysis.

2 Overview of the Tree-Traversal Orientation Analysis

The tree-traversal orientation analysis uses an optimizing compiler to automatically instrument a program at each dereference of a tree node. For modularity, this instrumentation is a single function call to a library, linked with the compiled code. Each memory access is classified and used to create a *orientation score* for the tree.

In the context of this work, a *tree* is any linked data structure where there is at most one path between any two nodes. A *traversal* is any method for visiting all the nodes of a tree.

In a *breadth-first traversal*, all of the nodes on one level of a tree are visited before the traversal moves to the next level. Any memory access deemed to be part of a breadth-first traversal is referred to as a *breadth-first access*. Breadth-first traversals are often implemented using queues that contain the nodes that still need to be visited. As a node is processed, its children are added to the queue of nodes to be processed, ensuring that one level of the tree is traversed in its entirety before any nodes on the next level are visited. Conversely, in a

depth-first traversal once a node is visited, one of its children is selected and the traversal continues from that child. Only when the subtree rooted by that child is fully processed is another child of the original node processed. Any memory access deemed to be part of a depth-first traversal is referred to as a *depth-first access*.

The instrumentation function updates an orientation score of the corresponding tree at each access. If the access is characterized as a breadth-first access, -1 is added to the orientation score. When a depth-first access occurs, $+1$ is added to the orientation score. Linked list traversals can be considered both depth- and breadth-first. To ensure that list traversals are identified, we classify accesses from a 1-ary node to its only child as depth-first [2]. A breadth-first traversal over a tree containing nodes with only one child will still appear breadth-first, as the traversal will still move among the nodes on every level of the tree, and not directly to the child of a 1-ary node. Finally, an access which cannot be classified as either breadth- or depth-first will force the score closer to 0 by either adding or subtracting 1 from the score. After all accesses have been analyzed, the orientation score is divided by the number of accesses, and returns a *traversal orientation score* in the range $[-1, 1]$ for each tree. A tree with orientation score of -1 is always traversed in breadth-first order. A score of $+1$ indicates pure depth-first traversal, and a score of 0 indicates that there is no predominant order of traversal.

3 Implementation of the Tree-Traversal Orientation Analysis

The implementation of this analysis has two primary objectives: (1) require no additional effort on the part of the programmer; (2) require minimal alterations to the compiler using the analysis. We modified the Open Research Compiler (ORC) [5] to instrument programs with calls to our Tree Analysis Library (TAL). The TAL performs all of the analysis, using the memory access information passed to it by the instrumented program. All of the computation required for the analysis is done within the instrumented code.

The TAL only analyzes trees that are represented by recursive data structures which are referenced by pointers and where pointers to children are in separate fields. For instance, TAL does not handle (1) trees where the children of a node are stored in a dynamically allocated array of pointers; and (2) trees in which the pointers to children are in fields that are unions or of type void.

In order to analyze the traversal orientation of trees, the compiler has to be able to identify the structures that represent trees in the first place. Fortunately, Ghiya and Hendren [10] have devised a static shape analysis that can determine if a pointer points to a tree data structure. Ghiya and Hendren's analysis would also allow the compiler to identify disjoint data structures using the interference matrix created by their algorithm.Ghiya and Hendren's analysis may prove

---

[2] Double-linked lists are considered 1-ary nodes because one of the pointers will be identified as a parent pointer.

conservative and not identify all possible tree-like data structures. Thus, the programmer can supplement Ghiya and Hendren's analysis with annotations. Either the interference matrix in Ghiya and Hendren's analysis or Lattner and Adve's Data Structure Analysis can be used to identify disjoint data structures [13]. Once Ghiya and Hendren's algorithm has been run, the symbol table can be annotated with the correct information as to whether or not a pointer points to a tree-like data structure. In the interest of time, for the experimental evaluation presented in this paper instead of implementing Ghiya and Hendren's analysis, we annotated the source code of the benchmarks with a special label to identify the data structures that represent trees.

Our modified version of the ORC adds instrumentation for each dereference to a tree structure, t, that calls the analysis function in the TAL with (1) the address of t, (2) all of the recursive fields of t linking node t to nodes $c_1 \ldots c_n$ and (3) a unique number to identify disjoint data structures. The TAL keeps a stack of choice-points for each tree. These choice-points are used to determine if a memory access can be considered part of a depth-first traversal. A choice-point structure contains the memory address of a node in the tree, t, and the memory addresses of all n children, $c_1 \ldots c_n$.

Let cp_{top} be the address of the choice-point on top of the choice-point stack; cp_{low} be the address of a choice point below cp_{top} in the choice-point stack; and t be athe current memory access to a tree node. t is considered depth-first if and only if:

1. cp_{top} is equal to t; OR
2. a child c_i of cp_{top} is equal to t; OR
3. there exists a lower choice point in the stack, cp_{low}, such that cp_{low} or a child of cp_{low} is equal t AND all of the choice-points on the stack above cp_{low} have been visited.

If case (2) identifies a depth-first access then a new choice-point, representing node t having all of the children $c_1 \ldots c_n$ of t, is pushed onto the stack. If a new choice-point is to be added to the stack, special care must be taken when copying the addresses of child pointers into the new choice-point. Tree structures often contain pointers to parent nodes. The addresses in these pointers must not be added to the new choice-point as children of the tree node t. To identify parent pointers the TAL compares the addresses of each possible child c_i in t, with the addresses stored in the choice points on the stack. If case (3) identifies a depth first access then all of the choice-points up to cp_{low} are popped off the stack and we leave the number of accesses and score unchanged.

To identify breadth-first searches the TAL maintains an *open list* and a *next list*. Both lists are implemented as double-ended bit vectors. When a memory access m occurs, if m is in the open list, then m is a breadth-first access. The children of m are added to the next list, and m is removed from the open list. When the open list is exhausted, the next list becomes the open list and the next list is emptied. The initial entry in the list is set at bit 0. Each bit in the bit vector represents a byte in the address space. Each new entry is added to the

vector based upon its distance from the initial entry. If the points-to analysis can identify that all of the objects pointed to by the pointer are of the same type, Lattner and Adve refer to this as being *type-homogeneous*, then size of the bit representation of the open list may be further reduced by using a bit to represent each structure instead of each byte.

A program may have more than one traversal of the same tree occurring simultaneously, or the orientation of traversal of a tree may change during execution. To deal with this situation, the TAL maintains a list of active data structures. This list contains multiple choice-point stacks and pairs of open and next lists. For instance, if a memory access m is not considered depth-first for any active choice-point stack, a new choice-point stack is created representing a depth-first traversal rooted at m. New choice-point stacks are created even if the access was classified as being breadth-first.

The data structures in the active list are either choice-point stacks or pairs of open/next lists. An access *matches* a choice-point stack if it is deemed to be a depth-first access by that choice-point stack. Conversely, the access matches a pair of open/next lists if it is deemed to be breadth first by those lists.

Whenever a memory access to a tree node occurs, this access is checked against all the data structures in the active list. A reference counter is associated with each data structure. If the access does not match a data structure, the reference counter of that structure is incremented. When the access matches a data structure, the data structure reference counter is reset to zero. If the reference counter of an structure reaches a threshold (16 in our implementation) the structure is removed from the active list and its space is reclaimed.

3.1 Time and Space Complexity

In the interest of readability, the analysis for the identification of depth-first and breadth-first accesses will be considered separately. In this analysis, we consider a tree containing V nodes.

First we consider the space and time complexity of identifying a depth-first memory access, assuming tree nodes have a minimum degree of b. The number of entries in a choice-point stack is bounded by $O(\log_b V)$. The bound on the number of nodes in the choice-point stack stems from the fact that before a choice-point representing node t is pushed onto the stack, the choice-point representing the parent of t is on the top of the stack. To consider the time complexity of determining if a memory access is a depth-first access we let cp_{top} be the node on the top of the choice-point stack. In the worst case, a memory access is not to cp_{top}, not to any children of cp_{top}, nor is it to an ancestor of cp_{top} and thus all entries in the stack have to be checked. It takes $O(b)$ operations to check the children of cp_{top}, and $O(\log_b V)$ operations to check all ancestors of cp_{top} on the stack. In addition, for an access, a new choice-point may be added, or several may be removed from the stack, which also is $O(b)$ or $O(\log_b V)$ depending on which of those values is larger. Thus, the worst case total time complexity is $O(b + \log_b V)$ per memory access.

In the breadth-first analysis, the size of a double-ended vector used is largely determined by the arrangement of the nodes in memory. We assume that nodes are located relatively close together in memory, and thus the size of the vector is not significant. This assumption was confirmed in the experimental results performed on the system. If the memory allocator packs the structures contiguously then the breadth-first traversal uses $O(V)$ memory because both the open list and the next list may grow proportionally to the number of nodes in the largest level of the tree data structure being profiled. A lookup in the double-ended vector takes $O(1)$ time. Insertions into the vector may require growing the vector, which will take time proportional to the new size of the vector.

Several active lists and stacks may be maintained during the analysis. The maximum value that the reference counters for the choice point stacks and open and next lists are allowed to grow to will limit the number of active structures in the system at any point in time. Thus, the maximum number of active structures can be considered constant, and is independent of the size of the tree. As a result, the fact that multiple lists and stacks are maintained does not affect the order complexity of the analysis.

4   Experimentation

To evaluate the accuracy of the analysis we ran a series of experiments on programs we designed to test the tree-orientation analysis as well as on the Olden benchmark suite. Our analysis was able to correctly identify the expected traversal orientation in all the benchmarks and required only 7% additional memory on average.

4.1   Experimental Setup

In order to determine both the accuracy of the tree-orientation analysis, and its use of both memory and time, tests were run on fifteen tree-based programs. We created a collection of programs to determine if the analysis correctly identifies standard tree traversals. In addition, we applied the tree-traversal orientation analysis to several of the Olden benchmarks.

For each program tested, the code was compiled both with and without the profiling code inserted. The maximum amount of memory used, and the execution times of both the instrumented and non-instrumented programs were calculated. The value(s) returned by the analysis inside the profiled code were also recorded. Using these values, it is possible to demonstrate the effectiveness of the analysis in properly determining tree traversals while demonstrating that the profiled code does not use significantly more memory than the original code.

All of the experiments were performed using the Open Research Compiler version 2.1. Instrumentation was added to the code by making minor modifications to the ORC to add function calls to the program wherever a tree pointer was dereferenced. The analysis library described in Section 3 was then linked with the instrumented code. The machine used for the experiments had a 1.3 GHz Itanium2 processor with 1 GB of RAM.

4.2 Programs with Known Traversal Orientation

Seven programs were written to test the effectiveness of the analysis. Each of these programs has specific properties that allowed the accuracy of the orientation analysis to be evaluated. In the context of this work, a *search* in a tree is a traversal that ends before exhausting all the nodes in the tree.

- `RandomDepth`: A binary tree is traversed using a depth-first oriented traversal. At each node, the order in which the children are visited is chosen at random.
- `BreadthFirst`: A breadth-first traversal is made over a binary tree.
- `DepthBreadth`: A depth-first traversal is performed over a binary tree, followed by a breadth-first traversal over the same tree.
- `NonStandard`: A non-standard traversal of a binary tree is performed. This traversal progresses in the following order: (1) data in the current node n and the right child of n are processed (2) the traversal is recursively performed on the left child of n; (3) the traversal is recursively performed on the right child of the right child of n; and (4) the traversal is recursively performed on the left child of the right child of n. This traversal would produce the following node access sequence for the tree of Figure 1: $\langle 1, 3, 2, 5, 4, 9, 8, 11, 10, 7, 15, 14, 6, 13, 12\rangle$. This traversal is neither depth- nor breadth-first, and should be recognized as such.
- `MultiDepth`: Several depth-first traversals of a binary tree are performed by this program, varying the point at which the data in each node is accessed. *i.e.* The data field(s) in the tree node are accessed either before the recursive calls to the traversal function, in between the recursive calls (if the structure has more then 1 child), or after the recursive calls to the children have been completed.
- `BreadthSearch`: A tree with a random branching factor for each node (with 2 to 6 children per node) is searched, breadth-first, several times.
- `BinarySearch`: Several tree searches are performed on a binary search tree.

4.3 Results of the Tree-Traversal Orientation Analysis

The tree-traversal orientation analysis was able to correctly identify the expected traversal orientation for all of the programs that we created as well as for all of the Olden benchmarks.

Table 1 gives the value computed by the tree-traversal orientation analysis for each of the programs created to test the analysis, along with the expected value. Each program has a single tree, thus Table 1 reports the score for the program.

The original Olden benchmarks were designed for multi-processor machines [4]. We used a version of the Olden benchmarks specifically converted for use with uniprocessor machines [15]. We used the following Olden benchmarks in our evaluation: BH, `Bisort`, `Health`, MST, `Perimeter`, `Power`, `Treeadd` and TSP [3]. Four of

---

[3] Benchmarks `em3d` and `Voronoi` are omitted from this study because they could not be correctly converted to use 64 bit pointers.

Synthetic	Analysis Result		Olden	Analysis Result	
Benchmark	Expected	Experimental	Benchmark	Expected	Experimental
`RandomDepth`	1.0	1.000000	`BH`	0.0	0.010266
`BreadthFirst`	-1.0	-0.999992	`Bisort`	0.0	-0.001014
`DepthBreadth`	0.0	0.000000	`Health`	1.0	0.807330
`NonStandard`	Close to 0.0	0.136381	`MST`	low positive	0.335852
`MultiDepth`	1.0	0.999974	`Perimeter`	low positive	0.195177
`BreadthSearch`	-1.0	-0.995669	`Power`	1.0	0.991617
`BinarySearch`	1.0	0.941225	`Treeadd`	1.0	1.000000
			`TSP`	low positive	0.173267

Table 1. Analysis results on the synthetic and Olden benchmarks.

the Olden benchmark programs, `Bisort`, `Perimeter`, `Power`, and `Treeadd`, use a single binary tree. `MST` does not use a proper tree; rather it uses linked lists to deal with collisions inside a hash table. `BH`, `Health`, and `TSP` use a mixture of trees and linked lists connected as a single structure. In these benchmarks, linked lists hang off the nodes of trees, and traversals frequently start with depth-first access to the tree nodes and continue with accesses to the linked lists.

We will examine each of the Olden benchmark programs in turn:

- `BH` contains tree nodes which contain both pointers to the children of a node, as well as pointers to nodes which are not considered children. As a result, even though the program performs several depth-first traversals of the tree, it cannot be classified as depth-first because many of the children of nodes are not traversed. A score of 0 is expected, as the traversal is not truly depth-first, and is definitely not breadth-first.
- `Bisort` performs two operations. A depth-first traversal of the tree is performed to manipulate the values stored in the tree. A merge operation routinely manipulates both children of a single node at once, which is a breadth-first way of accessing nodes. Other merge operations are neither breadth- nor depth-first. As we have competing breadth- and depth-first accesses, as well as many non-standard accesses, a score close to 0 is expected.
- `Health` consists of a relatively small 4-ary tree, where each node in that tree contains several linked lists. During simulations, a depth-first traversal of the tree is performed, however, at each node, the linked lists are traversed. As both linked-list and depth-first traversals are scored the same, a score close to one is expected.
- `MST` is based around hash tables. The tree data structure used is a linked list for chaining collisions in the hash table. Since `MST` performs many short linked-list traversals, a small positive value is expected.
- `Perimeter` uses a quad tree with a parent pointer to compute the perimeter of a region in an image. The program recursively traverses the tree, but, during the depth-first traversal, uses parent pointers to find adjacent nodes in the tree. This access pattern exhibits a mostly depth-first traversal strategy but the deviations from this pattern lead us to expect a score close to 0.

Synthetic	Memory Usage (kbytes)		Olden	Memory Usage (kbytes)	
Benchmark	Original	Instrumented	Benchmark	Original	Instrumented
RandomDepth	854 976	855 040	BH	3 520	3 520
BreadthFirst	424 928	441 328	Bisort	3 008	3 040
DepthBreadth	220 112	252 896	Health	8 448	8 624
NonStandard	420 800	420 912	MST	4 752	6 272
MultiDepth	529 344	652 288	Perimeter	6 064	6 528
BreadthSearch	30 192	47 408	Power	3 648	3 712
BinarySearch	224 192	224 320	Treeadd	3 008	3 008
			TSP	3 024	3 040

Table 2. The maximum amount of memory used in kilobytes.

- **Power** repeatedly traverses a binary tree that consists of a two different types of structures. The tree traversal is depth-first and implemented via recursion. We therefore expect the TAL to calculate a score close to 1 for this program.
- **Treeadd** performs a single depth-first traversal of a tree, while the program recursively computes the sum of the nodes. As this traversal is purely depth-first we expect a score close to 1.
- **TSP** combines a linked list and a tree into a single data structure. Each node has two pointers that indicate its position in a double-linked list, and two pointers pointing to children in a binary tree. Both list traversals and depth-first tree traversals are performed on this structure, however neither traversal visits all the children of a node (as all the children would comprise the two real children of the node, and the next and previous nodes in the list). The expected result is a very low positive score for the traversal.

4.4 Memory Usage

Despite the fact that the TAL creates and manipulates many data structures, it only increase memory usage by a modest amount. Code profiled with TAL requires, on average, 7% more memory than the original version of the program.

Table 2 compares the amount of memory used by the profiled and non-profiled versions of the code. In every case, the maximum amount of memory used by the program during its execution is given. The worst memory performance of the Olden benchmark comes from MST, where memory use increased by 32% during profiling. In contrast, several of the programs - BH, Power, Treeadd and TSP - only required about 1% more memory than the non-instrumented code.

4.5 Timing Results

Our instrumentation increases the runtime of the profiled code by an average of 5.9 times over the non-profiled code. Given that the instrumented version is typically executed once and offline, the additional runtime is acceptable.

Synthetic	Execution Time (seconds)		Olden	Execution Time (seconds)	
Benchmark	Original	Instrumented	Benchmark	Original	Instrumented
RandomDepth	0.90	10.72	BH	0.45	6.88
BreadthFirst	1.09	1.81	Bisort	0.03	1.01
DepthBreadth	1.33	7.84	Health	0.05	12.06
NonStandard	0.36	2.68	MST	5.71	11.42
MultiDepth	0.47	3.94	Perimeter	0.02	1.55
BreadthSearch	0.36	1.83	Power	1.35	1.60
BinarySearch	0.20	2.75	Treeadd	0.13	6.35
			TSP	0.01	1.40

Table 3. Execution time in seconds.

Table 3 shows the runtime overhead introduced by the offline tree-orientation analysis. For the test programs, the profiled code runs 6.7 times slower than the non-profiled code on average. Execution time of the Olden benchmarks increased by an average of 5.5 times during profiled runs.

Although the instrumented code takes longer to run, the performance degradation caused by profiling is acceptable for three reasons. First, this analysis can be run infrequently, for final compilations of code, or when major changes have been made to the data structures or the associated traversals. Second, the analysis can correctly determine traversal order in a relatively short period of program execution. The size of the tree considered is unimportant, as long as a representative traversal is performed on it. A proper selection of a representative input to profiling could help ensure that the analysis does not take too long. Finally, the analysis performs all the necessary calculations during the profiled run of the code. Additional work need not be performed by the compiler in order to determine the orientation of the tree traversal.

5 Related Work

5.1 Analysis of Pointer-based Structures

Ghiya and Hendren provide a context-sensitive inter-procedural shape analysis for C that identifies the shape of data structures [10]. Their analysis differentiates structures where the shape is a cyclic graph, a directed acyclic graph, or a tree. This work allows a compiler to automatically identify tree-like data structures and the pointers to those data structures, both of which are necessary for our profiling framework.

Lattner and Adve develop a framework that partitions distinct instances of heap-based data structures [13]. The partitioning is performed by allocating the disjoint structures into independent allocation pools. Their algorithm uses a context-sensitive unification-based pointer analysis to identify disjoint data structures for their memory allocator. Compile time incurred at most 3% additional overhead. Lattner and Adve's analysis can be used to identify disjoint instances of trees but does not provide a shape analysis.

5.2 Potential Clients of the Tree-Orientation Analysis

Prefetching Luk and Mowry propose three software prefetching schemes for recursive data structures [14]. The first prefetching scheme, known as greedy prefetching, prefetches all of the children of a structure when that structure is accessed. Greedy prefetching was implemented in the SUIF research compiler and obtained up to a 45% improvement in runtime on the Olden benchmarks. The prefetch distance is not adjustable with this technique and thus it is not able to hide the considerable memory access latency experienced in modern processors. To allow the compiler to control the prefetch distance, Luk and Mowry propose two other prefetching schemes, history-pointer prefetching and data-linearization prefetching. When the application is running, history-pointer prefetching records the access pattern via a history-pointer in each structure during the first tree traversal. Once all of the history pointers have been initialized, the system can use them to prefetch data. History-pointer prefetching was implemented by hand, and in spite of the runtime and space overhead, this technique can obtain up to a 40% speedup compared with greedy prefetching. Data-linearization prefetching allocates nodes with high access affinity near each other to increase spatial locality. Data-linearization prefetching is both implicit, through the use of spatial locality, and explicit, through the use of prefetches to incremental memory locations. Data-linearization prefetching was implemented by hand and resulted in up to a 18% speedup over greedy prefetching.

Cahoon and McKinley use compile-time data flow analysis to develop a software prefetching scheme for linked data structures in Java [2]. Traversals of linked data structures are identified through the use of a recurrent update to a pointer that is placed within a loop or recursive call related to the traversal of the linked data structure. A jump pointer, similar to Luk and Mowry's history pointer, is added to each structure that is updated recurrently and jump pointers are used for prefetching when the application is traversing the data structure. The prefetching scheme results in performance improvements as large as 48% but Cahoon and McKinley note that "... consistent improvements are difficult to obtain."

Our analysis could be used with Luk and Mowry's history-pointer prefetching or Cahoon and McKinley's jump-pointer prefetching to reduce the overhead of computing the history pointers and increase the benefit from prefetching. In both schemes, the first traversal of the tree is used to calculate the history pointer. Unless the structure is traversed many times this initial overhead may not be amortized out, and performance degradation could result. If the traversal pattern of the tree is known, a custom allocator could be used to linearize the subtrees that are allocated and prefetching could be performed in the same fashion as Luk and Mowry's data-linearization prefetching during the first tree traversal or used to set the jump pointers before they can be initialized by the first tree traversal. This would also allow hardware prefetchers, which are commonly found on modern processors, to retrieve the data and eliminate much of the latency caused by compulsory misses.

Modifying Data Layout Calder *et al.* present a framework that uses profiling to find the temporal relationship between objects and to modify the data placement of the objects to reduce the number of cache misses [3]. A profiling phase creates a temporal relationship graph between objects where edges connect those objects likely to be in the cache together. To reduce the number of conflict misses, objects with high temporal locality are placed in memory locations that will not be mapped to the same cache blocks. Field reordering is used to place objects in memory to maximize locality and reduce capacity misses by reducing the size of the working set. Finally, Calder *et al.* reduce compulsory misses by allowing blocks of data to be efficiently prefetched. These techniques are applied to both statically- and dynamically-allocated data. For dynamically-allocated data, the data placement is accomplished using a custom memory allocator that places allocated objects into a specific allocation bin based on information about the object traversal pattern. Experimentation showed that the data cache miss rate could be reduced by 24% on average, with some reductions as high as 74%.

Chilimbi, Hill and Larus describe techniques to improve the locality of linked data structures while reducing conflict misses, namely cache-conscious allocation and cache-conscious reorganization [7]. Two semi-automatic tools are created to allow the programmer to use cache-conscious allocation, `ccmalloc`, and cache-conscious reorganization, `ccmorph`, for their data structures. The main idea behind the tools are clustering, packing data with high affinity into a cache block, and coloring, using k-coloring to represent a k-way set-associative cache to reduce cache-conflicts. Their memory allocator, `ccmalloc`, takes a memory address of an element that is likely to be accessed at the same time as the newly allocated object and allocates them near one another in memory. The tree reorganizer, `ccmorph`, copies subtrees and lays them out linearly. After tree reorganization, any references to nodes in the tree must be updated. Chilimbi, Hill and Larus obtained speedups of 28-138% using their cache-conscious allocation techniques.

The work by Calder *et al.* and Chilimbi, Hill and Larus both aim to improve cache-hit rates by modifying where data is allocated. The profiling used by Calder *et al.* could be combined with our tree-traversal analysis to allow more information to be given to the compiler. A custom allocator could be created that can be given hints by the compiler based on information in the profile that was collected. The allocator could use Calder *et al.*'s or Chilimbi, Hill and Larus' techniques to arrange data to avoid cache conflicts while increasing the locality of tree nodes by allocating the nodes based on the profile information.

Structure Reorganization Truong, Bodin and Seznec use semi-automatic techniques to improve the locality of dynamically allocated data structures based on field reorganization and instance interleaving [19]. *Field reorganization* groups fields of a data structure that are referenced together into the same cache line, while *instance interleaving* groups identical fields of different instances of a structure into a common area in memory. They present a memory allocator, `ialloc`, that allocates structures, or chunks of structures, into arenas to increase locality.

Chilimbi, Davidson and Larus used field reordering and structure splitting to improve the behavior of structures that are larger then a cache line [6]. Field reordering groups the fields of a structure that are accessed together into sets which will fit into a cache line. Chilimbi, Davidson and Larus also group the fields of structures into hot (frequently accessed) and cold (infrequently accessed) fields. These techniques can increase the number of hot fields that can fit in the cache and they improved execution time by 6 - 18% over other co-allocation schemes by reducing cache miss rates by 10 - 27%.

It would be possible to combine Truong, Bodin and Seznec's `ialloc` memory allocator and the idea of allocation arenas with compiler technology to perform structure splitting similar to that performed by Zhao *et al.* [20], to apply this technique to recursive data-structures instead of arrays.

6 Conclusions

This work presents an analysis that accurately identifies the predominant traversal orientation of trees in a program. The analysis gives a floating point value for each tree, representing how close to a pure breadth- or depth-first orientation the traversal of that tree is. This value can be used by many client optimizations inside a compiler or may be used by programmers to improve the data structure layout. Most of the work is performed by a static library used to profile instrumented code which only slightly increases memory use. The tree-traversal orientation analysis requires no work on the part of the programmer, and requires only minor modifications to a compiler.

Acknowledgments

References

1. A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *25th Very Large Data Base (VLDB) Conference*, Edinburgh, Scotland, 1999.
2. B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *10th International Conference on Parallel Architectures and Compilation Techniques (PACT '01)*, pages 280–291, Los Alamitos, CA, USA, 2001. IEEE Computer Society.
3. B. Calder, K. Chandra, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, 1998.
4. M. Carlisle and A. Rogers. Olden web page. http://www.cs.princeton.edu/~mcc/olden.html.

5. S. Chan, Z. H. Du, R. Ju, and C. Y. Wu. Web page: Open research compiler - aurora. http://sourceforge.net/projects/ipf-orc.

6. T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming Language Design and Implementation*, pages 13–24, 1999.

7. T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming Language Design and Implementation*, pages 1–12, 1999.

8. T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 conference on Programming Language Design and Implementation*, page 1990209, Berlin, Germany, 2002.

9. S. Choi, N. Kohout, S. Pamnani, D. Kim, and D. Yeung. A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching. *ACM Transactions on Computer Systems*, 22(2):214–280, 2004.

10. R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Symposium on Principles of Programming Languages (POPL)*, pages 1–15, St. Petersburg, Florida, January 1996.

11. A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y. Chen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 577–588. VLDB Endowment, 2005.

12. M. Itzkowitz, B. J. N. Wylie, C. Aoki, and N. Kosche. Memory profiling using hardware counters. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003.

13. C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 129–142, 2005.

14. C. K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *ACM SIGOPS Operating Systems Review*, 1996.

15. C.K. Luk. Web resource: Uniprocessor olden tarball. http://www.cs.cmu.edu/~luk/software/olden_SGI.tar.gz.

16. R. Niewiadomski, J. N. Amaral, and R. C. Holte. A performance study of data layout techniques for improving data locality in refinement-based pathfinding. *ACM Journal of Experimental Algorithmics*, 9(1.4):1–28, October 2004.

17. E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 101–112, Portland, Oregon, Jan. 2002.

18. J. Rao and K. A. Ross. Making B+ Trees cache conscious in main memory. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 475–486, 2000.

19. D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. In *Seventh International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 322–329, 1998.

20. P. Zhao, S. Cui, Y. Gao, R. Silvera, and J. N. Amaral. *Forma*: A framework for safe automatic array reshaping. *ACM Transactions on Programming Languages and Systems*, To Appear.