Improving Pointer-Based Codes Through Cache-Conscious Data Placement

Trishul M. Chilimbi
James R. Larus
Mark D. Hill

Technical Report #1365

March 1998

Improving Pointer-Based Codes Through Cache-Conscious Data Placement

Trishul M. Chilimbi, James R. Larus, and Mark D. Hill
Computer Sciences Department
University of Wisconsin–Madison
1210 W. Dayton St.
Madison, Wisconsin 53706
{chilimbi, larus, markhill}@cs.wisc.edu

Abstract

Processor and memory technology trends show a continual increase in the cost of accessing main memory. Machine designers have tried to mitigate the effect of this trend through hardware and software prefetching, multiple levels of cache, non-blocking caches, dynamic instruction scheduling, speculative execution, etc.

These techniques, unfortunately, have only been partially successful for pointer-manipulating programs. This paper explores the complementary approach of redesigning and reorganizing data structures to improve cache locality. Pointer-based structures allow data to be placed in arbitrary locations in memory, and consequently in a cache. This freedom enables a programmer to improve performance by applying techniques such as clustering, compression, and coloring.

To reduce the cost and complexity of applying these techniques, this paper also presents two semi-automatic techniques for implementing cache-conscious data structures with minimal programmer effort. The first reorganizes tree-like data structures to improve locality. The second is a cache-conscious heap allocator. Our evaluations—with a tree microbenchmark, four Olden benchmarks, and two large applications—show that cache-conscious data structures, including those implemented by semi-automatic techniques, can produce large performance improvements and outperform hardware and software prefetching.

1 Introduction

Since 1980, microprocessor performance has improved 60% per year. Over the same period, memory access time decreased only 10% per year [35], which has produced a large processor-memory imbalance. Figure 1 shows that the resulting processor–memory gap grew at 45% per year. Processor memory caches are the ubiquitous hardware solution to this problem [50, 44]. In the beginning, a single level of cache sufficed, but the increasing imbalance now demands a memory hierarchy, which introduces a large

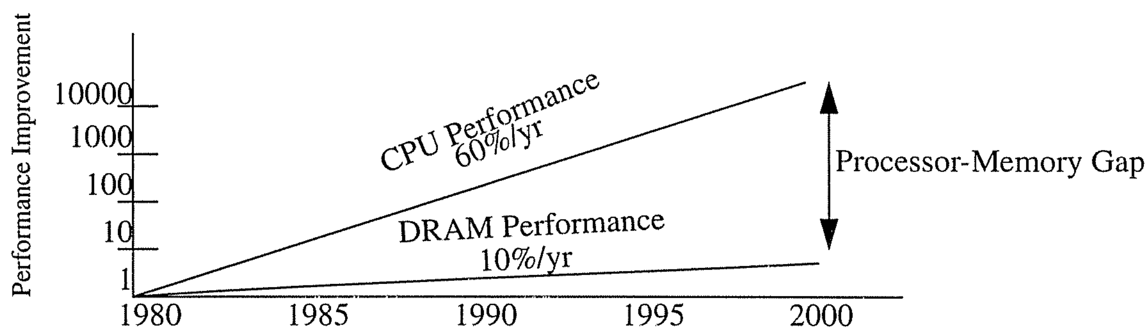

Figure 1. Processor-memory performance imbalance [35].

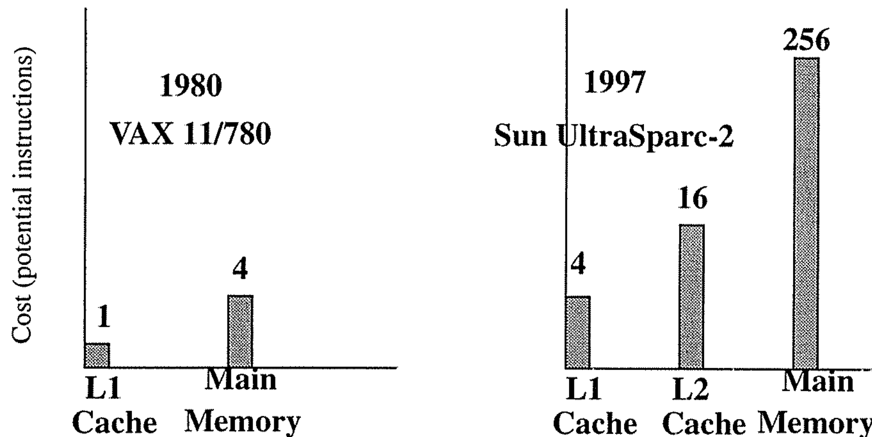

Figure 2. Memory hierarchy access costs.

disparity in memory-access costs. Figure 2 shows the opportunity cost (in potential instruction executions) of referencing data at various levels of a memory hierarchy. The 1980 cost (1–4) is for a VAX 11/780 [14], and the 1997 cost (1–256) is for an UltraSparc-2 [20]. The difference between a cache hit and miss is now almost two orders of magnitude. As a result, many programs' performance is dominated by memory references. Moreover, the large cost disparity undercuts the fundamental random-access memory (RAM) model used by most programmers to design data structures and algorithms.

Many hardware and software techniques—such as prefetching [32, 12], multithreading [28, 45], non-blocking caches [23], dynamic instruction scheduling, and speculative execution—have attempted to reduce or tolerate memory latency. These techniques require complex hardware and compilers, but have proven ineffective for many programs [10, 36].

The fundamental problem with these techniques is that they attack the manifestation (memory latency), not the source (poor reference locality), of the bottleneck. The only research that directly addressed this bottleneck has focused on improving cache locality in scientific programs that manipulate dense matrices [52, 13, 18]. Two properties of arrays underlie this work: uniform, random access to elements and a number theoretic basis for statically analyzing data dependencies.

Many data structures, however, contain pointers and share neither property. Pointer-based structures, fortunately, possess another, extremely useful property: elements in a compound data structure can be placed in different memory (and cache) locations without affecting a program's semantics. This paper describes techniques, useful in designing pointer-based data structures to exploit this *location-transparency property* to reduce memory access costs, and provides an analytic framework for these techniques. Applying the techniques to existing programs may require considerable effort. To reduce the barrier of programming effort and application understanding, this paper also presents two semi-automatic techniques for improving cache performance. Measurements demonstrate that cache-conscious data structures and data placement offer significant performance benefits—in most cases, outperforming state-of-the-art prefetching.

This paper makes the following contributions:

- ***Cache-conscious design techniques***—careful placement of structure elements in memory provides a programmer with a mechanism to improve the performance of a memory hierarchy. Three techniques—*clustering*, *compression*, and *coloring*—improve cache performance by increasing a data structure's spatial and temporal locality and by reducing cache conflicts. Clustering places structure elements likely to be accessed in succession in the same cache block. Compression reduces structure size or separates the active portion of structure elements. Coloring maps concurrently-accessed elements to non-conflicting regions of the cache. Section 2 discusses these techniques.
- ***Semi-automatic cache-conscious data placement***—Section 3 describes a transparent data reorganizer for tree-like structures (also lists and hash tables that use chaining), which applies the clustering and

Figure 3. Pointer-path access in a binary tree.

coloring techniques to the structure. A programmer only must supply a function to help traverse the data structure. In addition, the section describes a heap allocator that performs cache-conscious data placement at allocation time. In this case, a programmer only supplies an additional argument to *malloc*.

- ***Evaluation of cache-conscious data placement***—Section 4 presents measurements that demonstrate the performance benefits of cache-conscious data placement. Microbenchmarks show that cache-conscious trees outperform their naive counterparts by a factor of 4–5, and can even outperform B-trees by a factor of 1.5. We also applied our techniques to pointer-intensive programs in the Olden benchmark suite [39]. Semi-automatic cache-conscious data placement produced significant speedups, 28–194%, and even outperformed state-of-the-art prefetching by 3%–194%. In addition, we applied the techniques to full application programs. RADIANCE [49], a widely used ray-tracing program, ran 42% faster, and VIS [9], a model verification package, improved by 27%.
- ***Analytic framework***—Section 5 presents an analytic framework for evaluating and quantifying the performance benefits of cache-conscious pointer-based data structures. A key part of this framework is a data structure-centric cache model of a series of accesses that traverse a pointer-based data structure. The model characterizes the performance of a pointer-based data structure by its amortized miss rate over a sequence of pointer-path accesses. This paper applies the framework to cache-conscious trees and validates its predictions with microbenchmarks. The model has good predictive power. Our analytic framework and techniques work well for tree-like structures, lists, and certain kinds of hash tables. More complex graph-like structures, although difficult to analyze, still benefit from our cache-conscious data placement techniques.
- ***Differences between cache-memory and memory-disk performance gaps***—Appendix A illustrates subtle differences between the cache-memory and the memory-disk performance gaps, such as an optimally sized in-core B-tree node occupying 1.5 cache blocks, rather than a single cache block due to different access penalties.

2 Cache-Conscious Design Techniques

The section shows how to design and layout data structures for a memory organization that includes caches.

2.1 Cache Parameters

Caches can be parameterized by capacity, block size, associativity, indexing (virtual or physical), write hit/miss policy, and non-blocking degree. This paper focuses on three essential parameters: capacity, block size, and associativity. The other parameters are likely to have second-order effects (with the possible exception of non-blocking degree, which we did not consider as a design parameter in this paper).

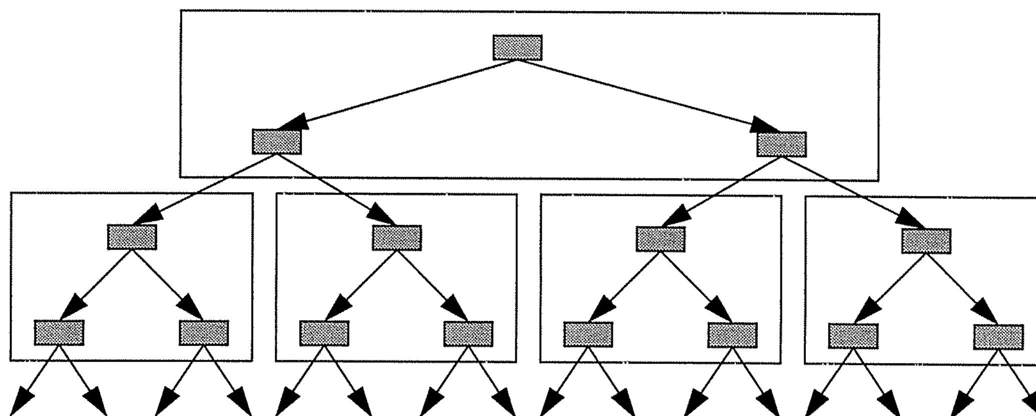

Figure 4. Subtree Clustering.

A *cache configuration C* is a vector < *c, b, a* > where:

c = cache capacity in sets

b = cache block size in words

a = cache associativity.

For example, a two-way set-associative 64 Kbyte cache with 64 byte blocks is <512, 16, 2> (assuming 4 byte words).

2.2 Design and Layout Techniques

This section discusses three general techniques—*clustering*, *compression*, and *coloring*—that can be combined in a wide variety of ways to produce cache-efficient data structures. The running example in this section is binary trees.

2.2.1 Clustering

Clustering tries to pack, in a cache block, data structure elements that are likely to be accessed successively. Clustering improves spatial and temporal locality and provides automatic prefetching.

An effective way to cluster a tree is to pack subtrees into a cache block. Figure 4 illustrates subtree clustering for a binary tree. An intuitive justification for binary subtree clustering is as follows (detailed analysis is in Section 5.3). For a series of random tree searches, the probability of accessing either child of a node is 1/2. With k nodes in a subtree clustered in a cache block, the expected number of accesses to the block is the height of the subtree, $log_2(k+1)$, which is larger than 2 for $k > 3$. An alternative is a depth-first clustering scheme, in which the k nodes in a block form a single parent-child-grandchild-... chain. In this case, the expected number of accesses to the block is:

$$1 + 1 \times \frac{1}{2} + 1 \times \frac{1}{2^2} + \ldots + 1 \times \frac{1}{2^{k-1}} = 2 \times \left(1 - \left(\frac{1}{2}\right)^k\right) \leq 2$$

Of course, this analysis assumes a random access pattern. For specific access patterns, such as depth-first search, other clustering schemes may be better. The disadvantage of subtree clustering is that tree modifications can destroy locality. However, for trees that change infrequently, clustering is effective.

2.2.2 Compression

Compressing data structure elements enables more elements to be clustered in a cache block. This both increases cache block utilization and shrinks a structure's memory footprint, which can reduce capacity and conflict misses. Compression typically requires additional processor operations to decode com-

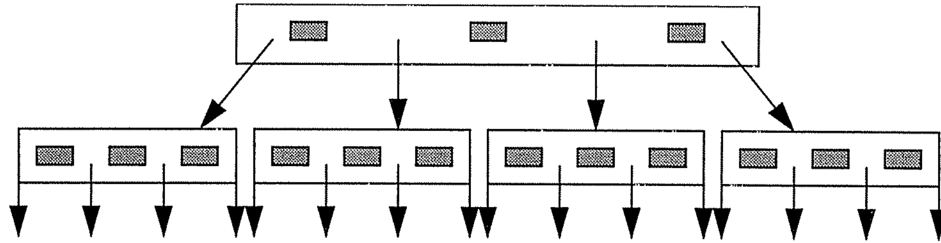

Figure 5. Compression by pointer elimination.

pressed information. However, with high memory access costs, computation may be cheaper than additional memory references. Structure compression techniques include data encodings such as key compression [15], and structure encodings such as pointer elimination and fluff extraction.

Pointer elimination replaces pointers by computed offsets. The classic example of pointer elimination is the implicit heap data structure, in which children of a node are stored at known offsets in an array. Another example is the tree structure in Figure 5, which eliminates the internal subtree pointers from the tree in Figure 4.

Fluff extraction is based on the observation that most searches examine only a portion of individual elements, until a match is found. Fluff extraction does not compress the data structure. Instead, it separates heavily accessed portions of data structure elements from rarely accessed portions (Figure 6). The heavily accessed portions can then be clustered to improve locality. Applying fluff extraction to the tree in Figure 5, results in an object that is structurally equivalent to a B-tree (except for the space in a B-tree node reserved for insertions).

2.2.3 Coloring

Caches have finite associativity, which means that only a limited number of concurrently accessed data elements can map to the same cache line without causing conflict misses. Coloring maps contemporaneously-accessed elements to non-conflicting regions of the cache. Figure 7 illustrates a two-color scheme for a 2-way set-associative cache for mapping data structure elements (easily extended to multiple colors). A cache with C cache sets (each set contains a = *associativity* blocks) is partitioned into two regions, one containing p sets, and the other C - p sets. Frequently accessed structure elements are uniquely mapped to the first cache region and the remaining elements are mapped to the other region. The mapping ensures that frequently accessed data structure elements do not conflict among themselves and are not replaced by infrequently accessed elements. For a tree structure, the p most frequently accessed elements are the top levels of the tree.

3 Semi-Automatic Cache-Conscious Data Placement

A programmer familiar with a machine's cache hierarchy can apply cache-conscious design to important structures. However, these techniques may require a detailed knowledge of a program's code and

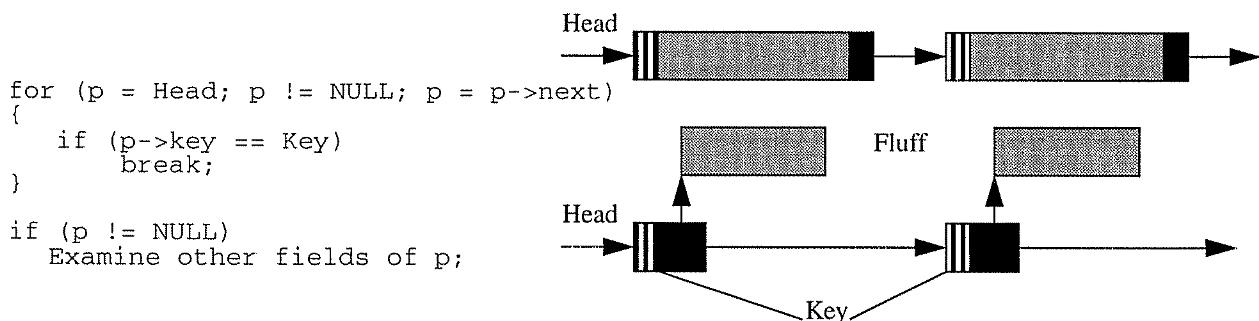

Figure 6. Compression through fluff extraction.

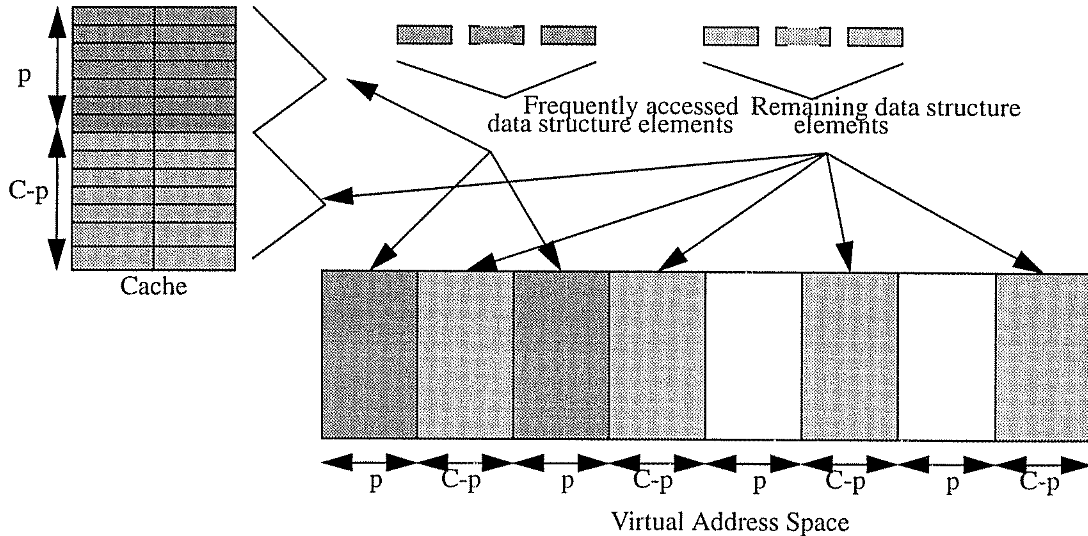

Figure 7. Coloring data structure elements to reduce cache conflicts.

data structures and considerable programming effort. This section describes two semi-automatic ways of applying cache-conscious data placement, both of which significantly reduce the required level of programming effort and knowledge and architectural familiarity. The first is a transparent (semantic-preserving) reorganizer for tree-like structures, and the second is a cache-conscious heap allocator.

3.1 Transparent Cache-Conscious Data Reorganization

In a language, such as C, that permits unrestricted use of pointers, analysis techniques are not yet powerful enough to distinguish all pointers to a structure. Without this knowledge, a system cannot move data structures without an application's cooperation (as it could in a language designed for garbage collection). However, if a programmer guarantees the safety of such an operation, a system can transparently reorganize a data structure to improve locality. To test this approach, we implemented a data reorganizer that applies the clustering and coloring (but not compression) techniques from Section 2.

This tool currently operates on tree-like structures (and lists) with homogeneous elements and without any external pointers into the middle of the structure (or any data structure that can be decomposed into components satisfying this property). In addition, it relaxes the strict tree property by allowing structure elements to contain a parent or predecessor pointer.

This type of reorganization is appropriate for read-mostly data structures, which are built in an

```
main()
{

  ......

  root = maketree(4096, ..., ...);

  ccreorganize(&root, next_node,
     Num_nodes, Max_kids, Cache_sets,
  Cache_blk_size, Cache_associativity,
                       Color_const);

  ......

}
```

```
Quadtree next_node(Quadtree node, int i)
{
  /* Valid values for i are -1,
                      1 ... Max_kids */
  switch(i) {
   case -1:
     return (node->parent);
   case 1:
     return (node->nw);
   case 2:
     return (node->ne);
   case 3:
     return (node->sw);
   case 4:
     return (node->se);
  }
}
```

Figure 8. Transparent cache-conscious data reorganization.

early phase of a computation, and subsequently heavily referenced. Neither the construction or consumption code need change, as the structure can be transparently reorganized between the two. For dynamically changing structures, the reorganization routine may have to be periodically invoked. A programmer supplies the data reorganization routine (which is templatized with respect to the structure type) with a pointer to the root of the data structure, a function to help traverse the structure, and cache parameters. Figure 8 contains the code used to reorganize the quadtree data structure of the Olden benchmark *perimeter.*

The reoganization routine copies the structure over to a contiguous block of memory. In the process it divides the tree-like structure into subtrees, starting from the root (see Figure 4), which are then laid out linearly. The structure is also colored such that the first p elements traversed map uniquely to a portion of the cache (determined by the *Color_const* parameter) and do not conflict with other structure elements (see Figure 7). The value of p and size of subtrees is determined by the cache parameters and structure element size. In addition, care is taken to ensure that the gaps in the virtual address space, required to implement coloring correspond to multiples of the virtual memory page size.

3.2 Cache-Conscious Heap Allocation

While the transparent data reorganization limits programming effort, it currently only works for tree-like structures that can be safely relocated. An alternative approach, which also requires little programmer intervention, is to apply cache-conscious data placement at allocation time, thereby eliminating the need to reorganize data.

Our cache-conscious heap allocator (`ccmalloc`) takes an additional parameter, which is a pointer to a data structure element that is likely to be in contemporaneous use (for e.g., the parent of a tree node). `ccmalloc` attempts to co-locate the new data item in the same physical cache block as the existing data. Figure 9 shows code from the Olden benchmark *health,* which illustrates its usage. In a memory hierarchy, different cache block sizes means that data can be co-located different ways. Our cache-conscious heap allocator attempts to co-locate in L2 cache blocks. In our system (Sun UltraSPARC 1), the L1 cache blocks are only 16 bytes (L2 are 64), which limits the number of objects that can fit in a block. Also, the bookkeeping overhead in the allocator is inversely proportional to the size of a cache block, so larger blocks are both more likely to be successful and incur less overhead.

An important issue is where to allocate the new data item if a cache block is full. For two reasons, our allocator tries to put the new data item as close to the existing pointer as possible. First, as the programmer has supplied a strong hint that the two items are likely to be accessed together, `ccmalloc` tries to co-locate them on the same page, to reduce the working set size. Second, putting them on the same page ensures they will not conflict in the cache. In addition to choosing a page, `ccmalloc` must select a block on the page. The *closest* strategy tries to allocate the new element in a cache block as close to the existing block as possible. The *new-block* strategy allocates the new data item in an unused cache block, optimistically reserving the reminder of the block for future co-location. The *first-fit* strategy uses a first-fit policy to find a cache block that has sufficient empty space. The next section evaluates these strategies.

```
void addList(struct List *list, struct Patient *patient) {
     struct List *b;
     while (list != NULL) {
        b = list;
        list = list->forward;
     }
     list = (struct List *) ccmalloc(sizeof(struct List), b);
     list->patient = patient;
     list->back = b;
     list->forward = NULL;
     b->forward = list;
}
```

Figure 9. Cache-conscious heap allocation.

4 Evaluation of Cache-Conscious Data Placement

To evaluate the benefit of the cache-conscious techniques, we use a combination of a simple microbenchmark, four small benchmarks from the Olden suite, and two large, real-world applications. The microbenchmark performed a large number of random searches on different forms of balanced trees. The Olden benchmarks were a variety of pointer-based applications written in C. The macrobenchmarks were a 60,000 line ray tracing program and a 160,000 line system that formally verifies finite state systems using Binary Decision Diagrams (BDDs).

4.1 Methodology

The microbenchmarks and macrobenchmarks were run on a Sun Ultraserver E5000, which contained 12 167Mhz UltraSPARC processors and 2 GB of memory running Solaris 2.5.1. This system has two levels of blocking cache — a < 1K, 4, 1> L1 data cache and a < 16K, 16, 1> L2 cache. A L1 data cache hit takes 1 cycle (i.e., $t_h = 1$). A L1 data cache miss, with a L2 cache hit, costs 6 additional cycles (i.e., $t_{mL1} = 6$). A L2 miss typically results in an additional 64 cycle delay (i.e., $t_{mL2} = 64$). All benchmarks were compiled with gcc (version 2.7.1) at the -O2 optimization level and run on a single processor of the E5000.

We also performed detailed cycle-by-cycle uniprocessor simulations of the four Olden benchmarks using RSIM [33]. RSIM is an execution driven simulator that models a dynamically-scheduled, out-of-order processor similar to the MIPS R10000. It simulates an aggressive memory hierarchy that includes a non-blocking, multiported and pipelined L1 cache, and a non-blocking and pipelined L2 cache. Table 1 contains the simulation parameters.

Table 1: Simulation Parameters.

Issue Width	4
Functional Units	2 Int, 2 FP, 2 Addr. gen., 1 Branch
Integer Multiply, Divide	3, 9 cycles
All Other Integer	1 cycle
FP Divide, Square Root	10, 10 cycles
All Other FP	3 cycles
Reorder Buffer Size	64
Branch Prediction Scheme	2-bit history counters
Branch Prediction Buffer Size	512
L1 Data Cache	16 KB, direct-mapped, dual ported, write-through
Write Buffer Size	8
L2 Cache	256 KB, 2-way set associative, write-back
Cache Line Size	128 bytes
L1 hit	1 cycle
L1 miss	9 cycles
L2 miss	60 cycles
MSHRs L1, L2	8, 8

4.2 Tree Microbenchmarks

This microbenchmark measures the performance of subtree-clustered binary trees (without internal pointer compression) that were colored to reduce cache conflicts. We call this structure a *transparent C-tree* and compare its performance against an in-core B-tree (see Appendix A), also colored to reduce cache conflicts, and against random and depth-first clustered binary trees. The microbenchmark does not involve insertions or deletions. The tree contained 2,097,151 keys and covered 40 MB, which is forty times the L2 cache's size. Since the L1 cache block size is 16 bytes and its capacity is 16K bytes, it provides practically no clustering or reuse, and hence its miss rate was very close to one. We measured the average search time for a randomly selected element, while varying the number of repeated searches to 1 million.

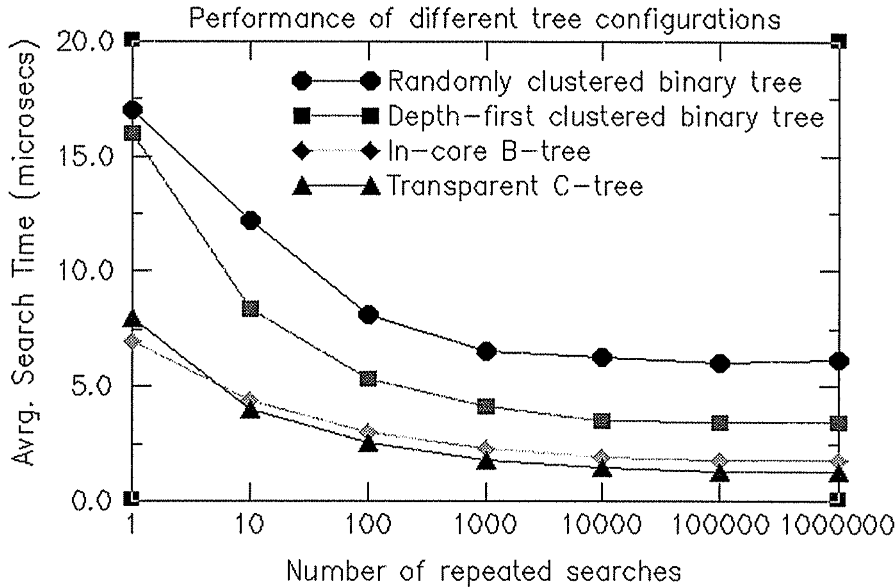

Figure 10. Binary tree microbenchmark.

Figure 10 shows that both B-trees and transparent C-trees outperform randomly clustered binary trees by up to a factor of 4–5, and depth-first clustered binary trees by up to a factor of 2.5–3. Moreover, transparent C-trees outperform B-trees by a factor of 1.5. The reason for this is that B-trees reserve extra space in tree nodes to handle insertion gracefully, and hence do not manage cache space as efficiently as transparent C-trees. However, we expect B-trees to perform better than transparent C-trees when trees change due to insertions and deletions.

4.3 Olden Benchmarks

Table 2 describes the four Olden benchmarks. We used RSIM to perform a detailed comparison of

Table 2: Benchmark characteristics.

Name	Description	Main Pointer-Based Structures	Input Data Set	Memory Allocated
TreeAdd	Sums the values stored in tree nodes	Binary tree	256 K nodes	4 MB
Health	Simulation of Columbian health care system	Doubly linked lists	max. level = 3, max. time =3000	828 KB
Mst	Computes minimum spanning tree of a graph	Array of singly linked lists	512 nodes	12 KB
Perimeter	Computes perimeter of regions in images	Quadtree	4K x 4K image	64 MB

the performance of our semi-automated cache-conscious data placement techniques—transparent clustering and coloring, and cache-conscious heap allocation (*closest*, *first-fit*, and *new-block* strategies)—against other latency reducing schemes, such as hardware prefetching (prefetching all loads and stores currently in the reorder buffer) and software prefetching (we implement Luk and Mowry's greedy prefetching scheme [29] by hand).

Figure 11 shows the results. The execution times are normalized against the original, unoptimized code. We used a commonly applied approach to attributing execution delays to various causes [34, 40]. If, in a cycle, the processor retires the maximum number of instructions, that cycle is counted as busy time. Otherwise, the cycle is charged to the stall time component corresponding to the first instruction that could

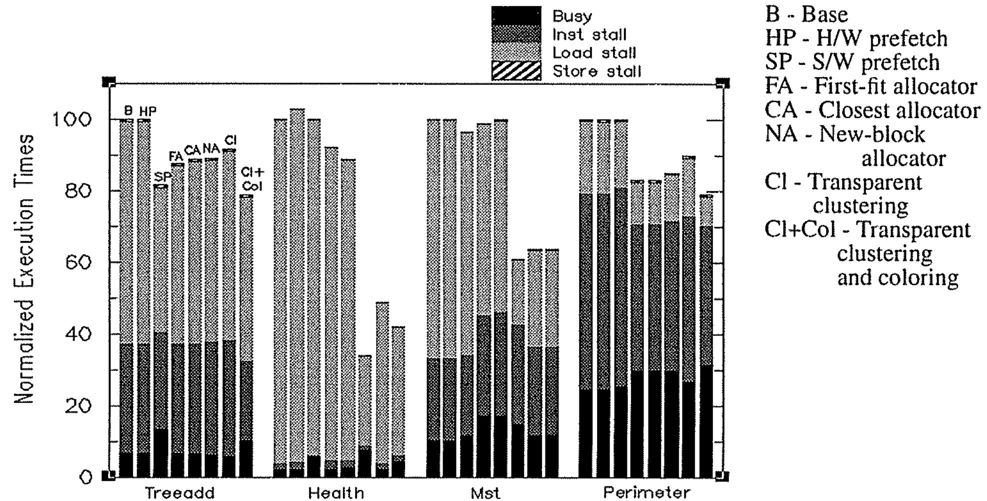

Figure 11. Performance of cache-conscious data placement.

not be retired.

Treeadd and *perimeter* both create their pointer-based structures (trees) at program start-up and do not subsequently modify them. Although cache-conscious data placement improves performance, the gain is only 10–20% because structure elements are created in the same order as the dominant traversal order, which produces a "natural" cache-conscious layout. However, all cache-conscious data placement techniques outperform hardware prefetching and are competitive with software prefetching for *treeadd*, and outperform both software and hardware prefetching for *perimeter*. The *new-block* allocation policy requires 12% and 30% more memory than *closest* and *first-fit* allocation policies, for *treeadd* and *perimeter* respectively (primarily due to leaf nodes being allocated in new blocks).

Health's primary data structure is linked lists, to which elements are repeatedly added and removed. The cache-conscious version periodically invoked the transparent clustering and coloring routine to reorganize the lists (no attempt was made to determine the optimal interval between invocations). Despite this overhead, transparent clustering and coloring significantly outperforming both software and hardware prefetching. Not surprisingly, the *new-block* allocation strategy, which left space in cache blocks to add new list elements, outperformed the other allocators, at a cost of 7% additional memory.

Mst's primary data structure is a hash table that uses chaining for collisions. It constructs this structure at program start-up and it does not change during program execution. As for *health*, the *new-block* allocator and transparent clustering and coloring, significantly outperformed other schemes. Coloring did not have much impact since the hash table lists were short. Since the hash lists are short, with no locality between lists, incorrect placement incurs a high penalty. The *new-block* allocator significantly outperformed both *first-fit*, and *closest* allocation schemes, at a cost of only 3% extra memory.

In summary, transparent clustering and coloring outperformed hardware and software prefetching schemes for all benchmarks, resulting in speedups of 28–138% over the base case, and 3–138% over prefetching. With the exception of *treeadd*, the *new-block* allocation strategy alone outperformed prefetching by 12–194%. In addition, the *new-block* allocator compares favorably with the other allocations schemes, with low memory overheads (with the exception of *perimeter*).

4.4 Macrobenchmarks

We also studied the impact of cache-conscious data placement on two real-world applications. RADIANCE is a tool for modeling the distribution of visible radiation in an illuminated space [49]. Its input is a three-dimensional geometric model of the space. Using radiosity equations and ray tracing, it produces a map of spectral radiance values in a color image. RADIANCE's primary data structure is an

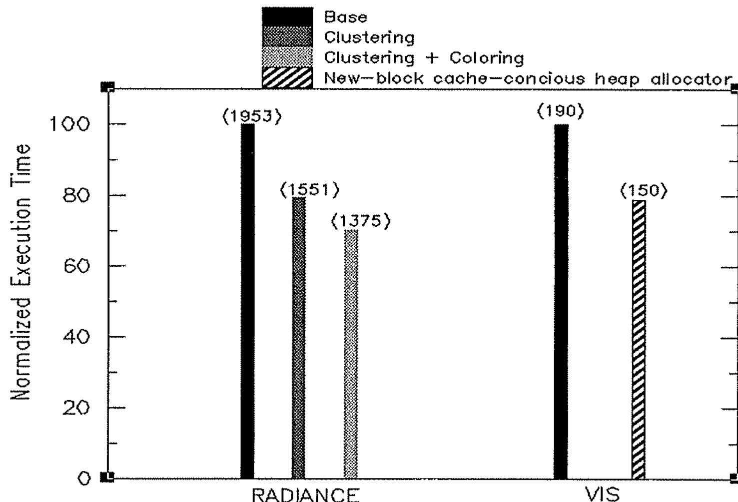

Figure 12. RADIANCE and VIS Applications. Actual execution times above each bar.

octree, which represents the scene to be modeled. This structure is highly optimized. The program uses implicit knowledge of the structure's layout to eliminate pointers, much like an implicit heap, and it lays out this structure in depth-first order (consequently, it did not make sense to use a cache-conscious heap allocator in this case). We changed the octree to use subtree clustering and colored the data structure to reduce cache conflicts.

VIS (Verification Interacting with Synthesis) is a system for formal verification, synthesis, and simulation of finite state systems [9]. VIS synthesizes finite state systems and/or verifies properties of these systems from Verilog descriptions. The fundamental data structure used in VIS is a multi-level network of latches and combinational gates, which is represented by Binary Decision Diagrams (BDDs). Since BDDs are DAGs, our transparent clustering and coloring technique is not directly applicable. However, we modified VIS to use our cache-conscious heap allocator with the *new-block* strategy.

Figure 12 shows the results. Cache-conscious clustering and coloring produced a speedup of 42% for RADIANCE, and cache-conscious heap allocation resulted in a speedup of 27% for VIS. The result for VIS demonstrates that cache-conscious data placement can even improve the performance of graph-like data structures, in which data elements have multiple parents. Significantly, very few changes to these 60–160 thousand line programs produced these large performance improvements.

4.5 Discussion

Table 3 summarizes the trade-offs among the cache-conscious data placement techniques. The techniques in this paper focus on single data structures. Real programs, of course, use multiple data structures, though often references to one structure predominates. Our techniques can be applied to each structure in turn to improve its performance. Future work will consider interactions among different structures.

Table 3: Summary of cache-conscious data placement techniques.

Technique	Data Structures	Program Knowledge	Source Code Modification	Suitability	Performance
CC Design	Universal	High	Large	All structures	High
Transparent Data Reorganization	Tree-like	Moderate	Small	Mostly-static trees, lists	Moderate–High
CC Heap Allocation	Universal	Low	Small	Dynamic structures	Moderate–High

5 Analytic Framework

Although the cache-conscious data placement techniques can improve a structure's spatial and temporal locality, their description is ad hoc. The framework presented in this section addresses this difficulty by quantifying their performance advantage. The framework permits *a priori* estimation of the benefits of these techniques. Its intended use is not to estimate the cache performance of a data structure, but rather to compare the relative performance of a structure with its cache-conscious counterpart.

A key part of the framework is a data structure-centric cache model that analyzes the behavior of a series of accesses that traverse pointer-paths in pointer-based data structures. A pointer-path access references multiple elements of a data structure by traversing pointers (Figure 3). Some examples are: searching for an element in a tree, or traversing a linked list. To make the details concrete, this paper applies the analytic framework to predict the steady-state performance of cache-conscious trees. The framework can also predict the start-up or transient performance of pointer-based data structures.

5.1 Analytic Model

For a two level blocking cache configuration, the expected memory access time for a pointer-path access to an in-core pointer-based data structure is given by

$t_{memory} = (t_h + m_{L1} \times t_{mL1} + m_{L1} \times m_{L2} \times t_{mL2}) \times (Memory\ References)$

t_h: level 1 cache access time

m_{L1}, m_{L2}: miss rates for the level 1 and level 2 caches respectively

t_{mL1}, t_{mL2}: miss penalties for the level 1 and level 2 caches respectively

A cache-conscious data structure should minimize this memory access time. Since miss penalties are determined by hardware, design and layout of a data structure can only attempt to minimize its miss rate. We now develop a simple model for computing a data structure's miss rate. Since a pointer-path access to a data structure can reference multiple structure elements, let $m(i)$ represent the miss rate for the i-th pointer-path access to the structure. Given a sequence of p pointer-path accesses to the structure, we define the amortized miss rate as

$$m_a(p) = \frac{\sum_{i=1}^{p} m(i)}{p} \tag{1}$$

For a long, random sequence of pointer-path accesses, this amortized miss rate can be shown to approach a steady-state value, m_s (in fact, the limit exists for all but the most pathological sequence of values for $m(i)$). We define the amortized steady-state miss rate, m_s as

$$m_s = \lim_{p \to \infty} m_a(p) \tag{2}$$

Equation 2 implies that m_s can be approximated by $m_a(p)$ for large enough p. Thus the memory system performance of a pointer-based data structure that is repeatedly accessed randomly, can be characterized by this steady-state, amortized miss rate.

We examine this amortized miss rate for a cache configuration $C< c, b, a >$, where c is the cache capacity in sets, b is the cache block size in words, and a is the cache associativity. Consider a pointer-based data structure consisting of n homogenous elements, subjected to a random sequence of pointer-path accesses of the same type. Let D be a pointer-path access function that represents the average number of unique references required to access an element of the structure. D depends on the data structure, and the type of pointer-path access (if the pointer-path accesses are not of the same type, D additionally depends

on the distribution of the different access types). For example, D is $log_2(n+1)$ for a key search on a balanced binary search tree. Let the size of an individual structure element be e. If $e < b$, then $\lfloor b/e \rfloor$ is the number of structure elements that fit in a cache block. Let M represent the mapping of data structure elements to memory locations. Let K represent the average number of structure elements residing in the same cache block that are required for the current pointer-path access. K depends on the pointer-path access function D, the mapping of data structure elements to memory locations M, and the size of individual data structure elements e relative to the cache block size b, and is a measure of a data structure's spatial locality for the pointer-path access function, D. From the definition of K it follows that

$$1 \leq K \leq \left\lfloor \frac{b}{e} \right\rfloor$$

Let R represent the number of elements of the data structure required for the current pointer-path access that are already present in the cache because of prior accesses. R depends on D, M, e, the number of data structure elements, n, and cache replacement which depends on c, b, and a. $R(i)$ is the number of elements brought into the cache by prior accesses that are reused during the i-th pointer-path access, and is a measure of a data structure's temporal locality for the pointer-path access function, D. From the definition of R it follows that

$$0 \leq R \leq min\left(D, \left\lfloor \frac{b \times c \times a}{e} \right\rfloor\right)$$

With these definitions, the miss rate for a single pointer-path access to a pointer-based data structure can be written as

$$m(i) = \text{(number of cache misses) / (total references)}$$

$$m(i) = \frac{\frac{D - R(i)}{K}}{D} = \frac{1 - \frac{R(i)}{D}}{K} \tag{3}$$

The reuse function $R(i)$ is highly dependent on i, for small values of i, because initially, a data structure suffers from cold start misses. However, one is often interested in the steady-state performance of a data structure once start-up misses are eliminated. If a data structure is colored to reduce cache conflicts (see Section 2.2.3), then $R(i)$ will attain a constant value R_s when this steady state is reached. Since D and K are both independent of i, the amortized steady-state miss rate m_s of a data structure can be approximated by its amortized miss rate $m_a(p)$, for a large, random sequence of pointer-path accesses p, all of the same type, as follows

$$m_s \approx m_a(p)\big|_{large\ p} = \frac{\sum_{i=1}^{p} m(i)}{p} \approx \frac{1 - \frac{R_s}{D}}{K} \tag{4}$$

Equation 3 can be used to analyze the transient start-up behavior of a pointer-based data structure, and Equation 4 to analyze its steady-state behavior.

5.2 Speedup Analysis

The cache model above shows that a pointer-based data structure's miss rate can be decreased in three ways—increasing K, increasing R, or decreasing D. Decreasing D is not always possible if the data structure has been optimized for a uniform cost memory system, while the cache-conscious design techniques increase K and R. On the other hand, K can be increased by intelligently *clustering* data structure

elements into cache blocks. K can also be increased by *compressing* data structure elements, which permits an increased clustering of elements. Techniques that increase K also increase R, since data structure *compression*, as well as smarter *clustering*, make more efficient use of the cache and increase the likelihood of a structure element being re-referenced before being replaced. In addition, R can be increased by judiciously *coloring* data structure elements to reduce cache conflicts.

We use the model to derive an equation for speedup in terms of cache miss rates that results from applying cache-conscious data placement techniques to a pointer-based data structure. This metric is desirable, as speedup is often more meaningful than cache miss rate, and is easier to measure.

$$\textit{Cache-conscious speedup} = (t_{memory})_{Naive} / (t_{memory})_{Cache\text{-}conscious}$$

When only the structure layout is changed, the number of memory references remains the same and the equation reduces to

$$Cache-conscious\ Speedup = \frac{(t_h + (m_{L1})_{Naive} \times t_{m_{L1}} + (m_{L1} \times m_{L2})_{Naive} \times t_{m_{L2}})}{(t_h + (m_{L1})_{CC} \times t_{m_{L1}} + (m_{L1} \times m_{L2})_{CC} \times t_{m_{L2}})}$$

In the worst case, with pointer-path accesses to a data structure that is laid out naively, $K = 1$ and $R = 0$ (i.e., each cache block contains a single element with no reuse from prior accesses) and the miss rate is 1. Then, we have

$$Best\ case\ Cache-conscious\ Speedup = \frac{(t_h + t_{m_{L1}} + t_{m_{L2}})}{(t_h + (m_{L1})_{CC} \times t_{m_{L1}} + (m_{L1} \times m_{L2})_{CC} \times t_{m_{L2}})} \quad (5)$$

5.3 Steady-State Performance Analysis

The cache-conscious functions, K and R, must be computed before applying the model. This section demonstrates how to calculate the steady-state performance of a cache-conscious tree.(see Section 4.2) subjected to a series of random key searches.

Consider a balanced, complete binary tree of n nodes. Let the size of a node be e words. If the cache block size is b words and $e < b$, up to $\lfloor b/e \rfloor$ nodes can be clustered in a cache block. Let subtrees of size $k = \lfloor b/e \rfloor$ nodes fit in a cache block. The tree is colored so the top $(c/2 \times \lfloor b/e \rfloor \times a)$ nodes of the tree map uniquely to the first $c/2$ sets of the cache with no conflicts and the remaining nodes of the tree map into next $c/2$ sets of the cache (other divisions of the cache are possible). If the number of tree searches is large, we can ignore the start-up behavior, and approximate the data structure's performance by its amortized steady-state miss rate (Equation 4).

Coloring subtree-clustered binary trees ensures that, in steady-state, the top $(c/2 \times \lfloor b/e \rfloor \times a)$ nodes are present in the cache. A binary tree search examines $log_2(n+1)$ nodes, and in the worst-case (random searches on a large tree approximate this), the first $log_2((c/2 \times \lfloor b/e \rfloor \times a)+1)$ nodes will hit in the cache, and the remaining nodes will miss. Since subtrees of size $k = \lfloor b/e \rfloor$ nodes are clustered in cache blocks, a single cache block transfer brings in $log_2(k+1)$ nodes that are needed for the current search. Hence, the steady-state miss rate for the structure is

$$m_s = \frac{(\log_2(n+1) - \log_2(c/2 \times k \times a + 1))/(\log_2(k+1))}{\log_2(n+1)} = \frac{1 - \dfrac{\log_2(c/2 \times k \times a + 1)}{\log_2(n+1)}}{\log_2(k+1)}$$

Comparing with Equation 4, we get $K = log_2(k+1)$ and $R_s = log_2(c/2 \times k \times a + 1)$. This result indicates that cache-conscious trees have logarithmic spatial and temporal locality functions, which intuitively appear to be the best attainable, since the access function itself is logarithmic.

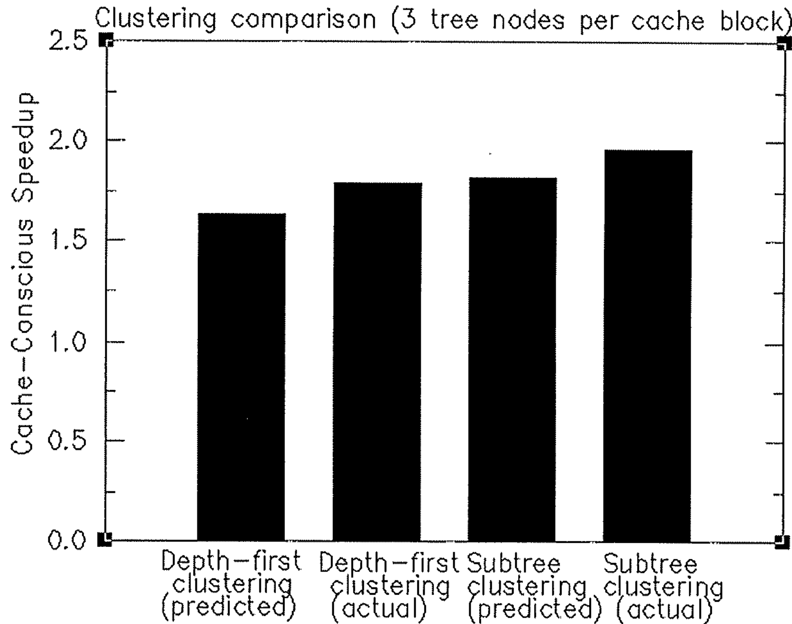

Figure 13. Predicted and actual effects of clustering.

5.4 Model Validation

This section validates the model in two ways. First it shows that the individual techniques of cache-conscious design and layout correspond to the model. Second, it validates the model's predictions of performance improvement. The model has good predictive power, underestimating the actual performance improvement by not more than 15% and accurately predicting the shape of speedup curves. Some reasons for this systematic underestimation might be a lower L1 cache miss rate (assumed 1 here) and TLB performance improvements not captured by our model.

The experimental setup is the same as before (see Section 4.1). The tree microbenchmark is used for the experiments of 1 million repeated searches for randomly generated keys in a tree containing 2,097,151 keys with 3 tree nodes in a cache block.

5.4.1 Clustering

First, we used the model to compare the performance benefits of subtree and depth-first clustering of trees and validated its predictions against real executions. In both cases, tree nodes were not colored to reduce cache conflicts, so all performance improvement is due to clustering. As noted previously, the L1 cache miss rate for this large tree is likely to be very close to 1. The L2 miss rate for the subtree clustered tree is $1/log_2(3+1) = 0.5$. The L2 miss rate for the depth-first clustered tree is $1/(2(1-0.125)) = 0.571$. Using these miss rates in Equation 5 for best-case cache-conscious speedup, we obtained the predictions in Figure 13. The model underestimates the speedup for both clustering techniques by only 8–9%.

5.4.2 Coloring

To validate the model's prediction of the benefit of coloring, we varied the number of tree nodes that are uniquely mapped to a region of the cache from 384 (one 8K page's worth) to 64 x 384 (half the L2 cache capacity). Although colored, tree nodes were not clustered, so the performance benefits are attributable to the coloring. The model predicts the L2 miss rate to be 1 - (log_2(nodes uniquely mapped+1)/ log_2(total nodes+1)). Figure 14 shows that the model underestimates the improvement by only 8–14% and accurately predicts the shape of the speedup curve.

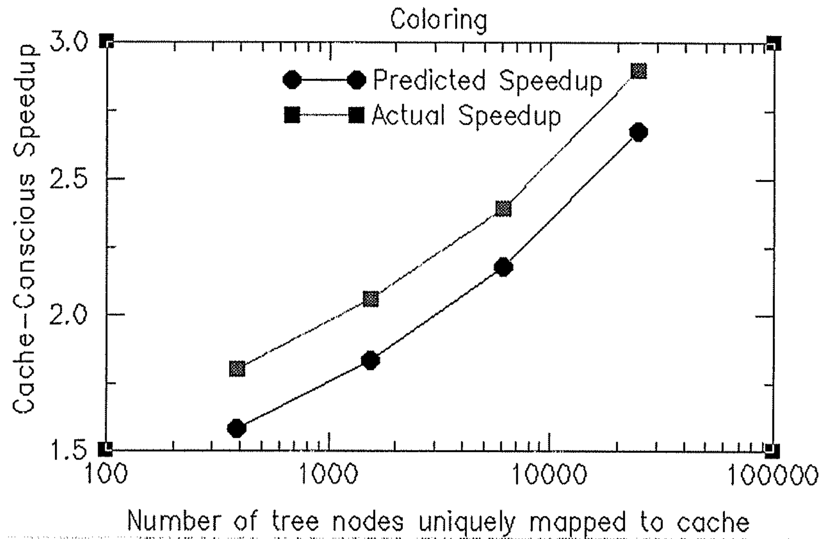

Figure 14. Predicted and actual effects of coloring.

5.4.3 Transparent C-trees

We apply the model to predict the performance advantage of transparent C-trees, which use both subtree clustering and coloring. For the experiments, subtrees of size 3 were clustered in a single cache block and 64 x 384 tree nodes (half the L2 cache capacity as 384 nodes fit in a 8K page) were colored into a unique portion of the L2 cache. The tree size was also increased from 262,144 to 4,194,304 nodes. The results are shown in Figure 15. As the graph shows, the model underestimated the actual speedup by only 15% and accurately predicted the shape of the curve.

6 Related Work

Previous research has attacked the processor-memory gap by reordering computations to increase their spatial and temporal locality [18, 52, 13]. Most of this work focused on codes that access arrays in a regular manner. Gannon et al. [18] studied an exhaustive approach that generated all possible permutations of a loop nest and selected the best one using an evaluation function. Wolf and Lam [52] developed a loop

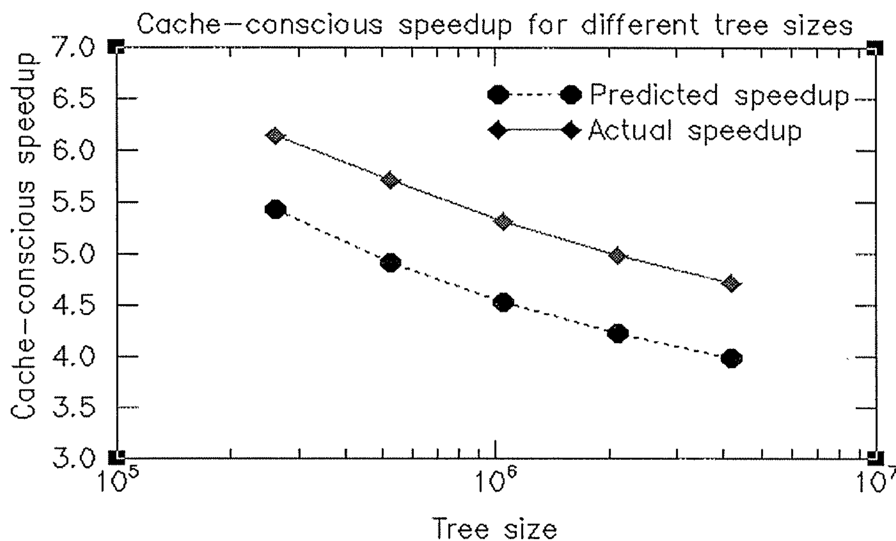

Figure 15. Predicted and actual speedup for C-trees.

transformation theory, based on unimodular matrix transformations, that unifies loop transforms such as interchange, reversal, and skewing. They used a heuristic algorithm to select the best combination of loop transformations. Carr et. al [13] used a simple model of spatial and temporal reuse of cache lines to apply compound loop transformations. We, on the other hand, consider an entirely different class of data structures. Pointer-based data structures do not support random access, and hence changing most programs' access patterns is impractical.

Database researchers long ago faced a similar performance gap between main memory and disk speeds. They designed specialized data structures, such as B-trees [7, 15], to bridge this gap. In addition, databases use clustering [6, 48, 17, 8] and compression [15] to improve a program's virtual memory performance. Appendix A shows that the spirit of database techniques carries over to in-core data structures, but different costs lead to different design decisions.

Clustering has also been used to improve virtual memory performance of Smalltalk and LISP systems [31, 47, 51, 24, 16] by reorganizing data structures during garbage collection. These studies focused on a program's paging behavior, not its cache behavior. Our work differs, not only because of the different cost for a cache miss and a page fault, but also because cache blocks are far smaller than memory pages.

Recently Calder et al. [11] have applied placement techniques developed for instruction caches [19, 37, 30] to data. Their technique reduces cache conflicts, and is based on profiling and assumes no prior knowledge of the data structure. Our work complements theirs by providing techniques and analysis for designing and laying out data structures, and semi-automatic techniques that improve a structure's cache locality without profiling.

Several other cost models have tried to capture the hierarchical nature of memory systems. The Uniform Memory Hierarchy (UMH) model of Alpern et al. [5] models memory as a sequence of increasingly large modules $<M_0, M_1, \ldots>$, in which each module M_u, is represented with 3 parameters, $<s_u, n_u, l_u>$. Intuitively, M_u is a box that holds n_u blocks, each of size s_u, and l_u is the latency for transferring this block to the next level of the hierarchy. The UMH model assumes that the ratio of n_u to s_u is the same for all modules, the ratio of s_u to s_{u-1} is a constant, and that the transfer cost between levels of the hierarchy can be represented by one function, $f(u)$. This model is closely related to the Hierarchical Memory Model (HMM) [2], and the Block Transfer model (BT) [3]. Each model is a family of machines parameterized by a function that represents the cost of accessing data. An HMM$f(x)$ is a RAM machine where referencing the k-th memory location costs $f(k)$. For a BT$f(x)$ machine, referencing the k-th memory location costs $f(k)$ as well. However, a block of length l starting at location k can be transferred at cost $f(k) + l$. The HMM model does not take spatial locality into account and, like the BT model, only permits one data transfer at a time, whereas the UMH model allows separate data blocks to be transferred simultaneously between different memory modules. To date, these models have only been applied to problems, such as matrix multiplications and FFT, in which the computation is oblivious to data values. Our model is more limited in scope and focuses on the cache behavior of in-core, pointer-based data structures.

Researchers have also used empirical models of program behavior [4, 41, 46] to analyze cache performance [38, 43, 21]. These efforts tailor the analysis to specific cache parameters, which limits their scope. Two exceptions are Agarwal's comprehensive cache model [1] and Singh's model [42]. Agarwal's model uses a large number of parameters, some of which appear to require measurements to calibrate. He provides performance validation that shows that the model's predictions are quite accurate. However, the model's complexity and large number of parameters, makes it difficult to gain insight into the impact of different cache parameters on performance. Singh presents a technique for calculating the cache miss rate for fully associative caches from a mathematical model of the behavior of workloads. His technique requires fewer parameters than Agarwal's model, but again measurements appear necessary to calibrate them. The model's predictions are accurate for large, fully associative caches, but are not as good for small caches. Hill [22] proposed the simple 3C model, which classifies cache misses into three categories—compulsory, capacity, and conflict. The model provides an intuitive explanation for the causes of cache misses, but it lacks predictive power. These cache models focus on analyzing and predicting a program's cache

performance, while we focus on the cache performance of individual memory-resident pointer-based data structures.

Lam et al. [25] developed a theoretical model of data conflicts in the cache and analyzed the implications for blocked array algorithms. They showed that cache interference is highly sensitive to the stride of data accesses and the size of blocks, which can result in wide variation in performance for different matrix sizes. Their cache model captures loop nests that access arrays in a regular manner, while our model focuses on series of pointer-path accesses to memory-resident pointer-based data structures.

LaMarca and Ladner [26, 27] considered the effects of caches on sorting algorithms, and improved performance by restructuring these algorithms to exploit caches. In addition, they constructed a cache-conscious heap structure that clustered and aligned heap elements to cache blocks. Their "collective analysis" models an algorithm's behavior in the presence of direct-mapped caches and obtains fairly accurate predictions. Their framework relies on the "independence reference assumption" [4] (memory references are independent), and is algorithm-centric, whereas ours is data structure-centric, and specifically targets correlations between multiple accesses to the same data structure.

7 Conclusions

Traditionally, in-core pointer-based data structures were designed and programmed as if memory access costs were uniform. Increasingly expensive memory hierarchies open an opportunity to achieve significant performance improvements by redesigning data structures to use caches more effectively. This paper demonstrates that three techniques—*clustering*, *compression*, and *coloring*—can improve the spatial and temporal locality of pointer-based data structures. Programmers can use the framework and cache-conscious techniques described in this paper to design key structures for new applications.

However, applying the techniques to existing codes may require considerable effort. This paper also shows how cache-conscious layout techniques can be semi-automatically applied. Our structure reorganizer and cache-conscious memory allocator greatly reduce the programming effort and application knowledge required.

Microbenchmarks show that cache-conscious trees outperform their naive counterparts by a factor of 4–5, and even outperform more-complex B-trees by a factor of 1.5. For pointer-based codes from the Olden benchmark suite, our semi-automatic cache-conscious data placement techniques result in significant speedups (28–194%) and even outperform state-of-the-art prefetching. When applied to two large, real-world applications, they produced speedups of 42% and 27%.

References

[1] A. Agarwal, M. Horowitz, and J. Hennessy. "An analytical cache model." *ACM Transactions on Computer Systems*, 7(2):184–215, 1989.

[2] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. "A model for hierarchical memory." In *Proceedings of the 19th Symposium on Theory of Computation*, pages 305–314, May 1987.

[3] A. Aggarwal, A. K. Chandra, and M. Snir. "Hierarchical memory with block transfer." In *Proceedings of the 28th Symposium on Foundations of Computer Science*, pages 204–216, Oct. 1987.

[4] A. V. Aho, P. J. Denning, and J. D. Ullman. "Principles of optimal page replacement." *Journal of the ACM*, 18(1):80–93, 1971.

[5] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. "The uniform memory hierarchy model of computation." *Submitted for publication*, 1992.

[6] J. Banerjee, W. Kim, and J. F. Garza. "Clustering a DAG for CAD databases." *IEEE Transactions on Software Engineering*, 14(11):1684–1699, 1988.

[7] R. Bayer and C. McCreight. "Organization and maintainence of large ordered indexes." *Acta Informatica*, 1(3):173–189, 1972.

[8] Veronique Benzaken and Claude Delobel. "Enhancing performance in a persistent object store: Clustering strategies in O2." In *Technical Report 50-90, Altair*, Aug. 1990.

[9] R. K. Brayton, G. D. Hachtel, A. S. Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto,

A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. R. Shilpe, G. Swamy, and T. Villa. "VIS: a system for verification and synthesis." In *Proceedings of the Eight International Conference on Computer Aided Verification*, July 1996.

[10] Doug Burger, James R. Goodman, and Alain Kagi. "Memory bandwidth limitations of future microprocessors." In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 78–89, May 1996.

[11] Brad Calder, Simmi John, and Todd Austin. "Cache-conscious data placement." In *University of California, San Diego, Technical Report*, Nov. 1997.

[12] David Callahan, Ken Kennedy, and Allan Poterfield. "Software prefetching." In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 40–52, April 1991.

[13] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. "Compiler optimizations for improving data locality." In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 252–262, Oct. 1994.

[14] Douglas W. Clark. "Cache performance in the VAX 11/780." *ACM Transactions on Computer Systems*, 1(1):24–37, 1983.

[15] Douglas Comer. "The ubiquitous B-tree." *ACM Computing Surveys*, 11(2):121–137, 1979.

[16] R. Courts. "Improving locality of reference in a garbage-collecting memory management system." *Communications of the ACM*, 31(9):1128–1138, 1988.

[17] P. Drew and R. King. "The performance and utility of the Cactis implementation algorithms." In *Proceedings of the 16th VLDB Conference*, pages 135–147, 1990.

[18] Dennis Gannon, William Jalby, and K. Gallivan. "Strategies for cache and local memory management by global program transformation." *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[19] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. "Procedure placement using temporal ordering information." In *Proceedings of MICRO-30*, Dec. 1997.

[20] Linley Gwennap. "Intel's P6 uses decoupled superscalar design." *Microprocessor Report*, 9(2):9–15, Oct. 16 1996.

[21] I. J. Haikala. "Cache hit ratios with geometric task switch intervals." In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 364–371, June 1984.

[22] Mark D. Hill and Alan Jay Smith. "Evaluating associativity in CPU caches." *IEEE Transactions on Computers*, C-38(12):1612–1630, December 1989.

[23] David Kroft. "Lockup-free instruction fetch/prefetch cache organization." In *The 8th Annual International Symposium on Computer Architecture*, pages 81–87, May 1981.

[24] M. S. Lam, P. R. Wilson, and T. G. Moher. "Object type directed garbage collection to improve locality." In *Proceedings of the International Workshop on Memory Management*, pages 16–18, Sept. 1992.

[25] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. "The cache performance and optimizations of blocked algorithms." In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, California, 1991.

[26] Anthony LaMarca and Richard E. Ladner. "The influence of caches on the performance of heaps." *ACM Journal of Experimental Algorithmics*, 1, 1996.

[27] Anthony LaMarca and Richard E. Ladner. "The influence of caches on the performance of sorting." In *Eight Annual ACM-SIAM Symposium on Discrete Algorithms*, Jan. 1997.

[28] James Laudon, Anoop Gupta, and Mark Horowitz. "Interleaving: A multithreading technique targeting multiprocessors and workstations." In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, San Jose, California, 1994.

[29] Chi-Keung Luk and Todd C. Mowry. "Compiler-based prefetching for recursive data structures." In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 222–233, Oct. 1996.

[30] Scott McFarling. "Program optimization for instruction caches." In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, 1989.

[31] D. A. Moon. "Garbage collection in a large LISP system." In *Conference Record of the 1984 Symposium on LISP and Functional Programming*, pages 235–246, Aug. 1984.

[32] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. "Design and evaluation of a compiler algorithm for prefetching." In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 62–73, October 1992.

[33] V. S. Pai, P. Ranganathan, and S. V. Adve. "RSIM reference manual version 1.0." In *Technical Report 9705, Dept. of Electrical and Computer Engineering, Rice University*, Aug. 1997.

[34] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. "An evaluation of memory consistency models for shared-memory systems with ILP processors." In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 12–23, Oct. 1996.

[35] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keaton, Christoforos Kazyrakis, Randi Thomas, and Katherine Yellick. "A case for intelligent RAM." In *IEEE Micro*, pages 34–44, Apr 1997.

[36] Sharon E. Perl and Richard L. Sites. "Studies of Windows NT performance using dynamic execution traces." In *Second*

Symposium on Operating Systems Design and Implementation, Oct. 1996.

[37] Karl Pettis and Robert C. Hansen. "Profile guided code positioning." *SIGPLAN Notices*, 25(6):16–27, June 1990. *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation.*

[38] G. S. Rao. "Performance analysis of cache memories." *Journal of the ACM*, 25(3):378–395, 1978.

[39] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. "Supporting dynamic data structures on distributed memory machines." *ACM Transactions on Programming Languages and Systems*, 17(2), 1995.

[40] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. "The impact of architectural trends on operating system performance." In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 285–298, Dec. 1995.

[41] J. H. Saltzer. "A simple linear model of demand paging performance." *Communications of the ACM*, 17(4):181–186, 1974.

[42] Jaswinder Pal Singh, Harold S. Stone, and Dominique F. Thiebaut. "A model of workloads and its use in miss-rate prediction for fully associative caches." *IEEE Transactions on Computers*, 41(7):811–825, 1992.

[43] A. J. Smith. "A comparative study of set associative memory mapping algorithms and their use for cache and main memory." *IEEE Trans. on Software Engineering*, 4(2):121–130, 1978.

[44] Alan J. Smith. "Cache memories." *ACM Computing Surveys*, 14(3):473–530, 1982.

[45] Burton J. Smith. "Architecture and applications of the HEP multiprocessor computer system." In *Real-Time Signal Processing IV*, pages 241–248, 1981.

[46] J. R. Spirn, editor. *Program Behavior: Models and Measurements*. Operating and Programming System Series, Elsevier, New York, 1977.

[47] J. W. Stamos. "Static grouping of small objects to enhance performance of a paged virtual memory." *ACM Transactions on Programming Languages and Systems*, 2(2):155–180, 1984.

[48] M. N. Tsangaris and J. Naughton. "On the performance of object clustering techniques." In *Proceedings of the 1992 ACM SIGMOD Intl. Conf. on Management of Data*, pages 144–153, June 1992.

[49] G. J. Ward. "The RADIANCE lighting simulation and rendering system." In *Proceedings of SIGGRAPH '94*, July 1994.

[50] M. V. Wilkes. "Slave memories and dynamic storage allocation." In *IEEE Trans. on Electronic Computers*, pages 270–271, April 1965.

[51] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. "Effective "static-graph" reorganization to improve locality in garbage-collected systems." *SIGPLAN Notices*, 26(6):177–191, June 1991. *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation.*

[52] Michael E. Wolf and Monica S. Lam. "A data locality optimizing algorithm." *SIGPLAN Notices*, 26(6):30–44, June 1991. *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation.*

Appendix A. B-Tree Node Sizing

B-trees are a structure that addresses the performance gap between memory and disk. The previous discussion showed that clustering and compressing a binary tree produces a structural similar to a B-tree. However, in order to efficiently handle insertions and deletions, B-trees reserve extra space in tree nodes. Due to the high cost of transferring data between disk and memory, B-trees nodes are sized to fit in a disk page when completely full. Thus it seems advantageous to size an in-core B-tree node to fit in a cache block.

We ran a microbenchmark to verify this hypothesis. The microbenchmark constructed an in-core B-tree containing 32767 keys, where each node except for the root contained between d and $2d$ keys. It then performed a million searches for randomly selected keys in this in-core B-tree. We performed a number of experiments varying the size of d and obtained the results shown in the graph (Figure 16).

The graph advocates a 96 byte in-core B-tree node when the cache block size is only 64 bytes. This seemingly surprising result, is intuitive on closer examination. A 96 byte in-core B-tree node implies that the node may contain 5 to 10 keys, whereas a 64 byte in-core B-tree node can contain only 3 to 6 keys. Since the cache block size is 64 bytes, in-core B-tree nodes that contain 5 or 6 keys fit in a single cache block, and those that contain 7 to 10 keys require 2 cache blocks. Since most B-tree nodes are not completely full, a 96 byte tree node may fit in a cache block. Even if the tree node occupies two cache blocks, the child pointer traversed during a search may reside in the first cache block. A larger in-core B-tree node causes less frequent node splitting, reducing the height of the tree and resulting in fewer accesses per search. Thus, this result indicates that for the optimal-sized in-core B-tree node, most node accesses require only a single cache block transfer. The occasional additional cache block required, is more than compensated for by the reduced tree height a larger node size entails. Hence we should size an in-core B-tree node such that

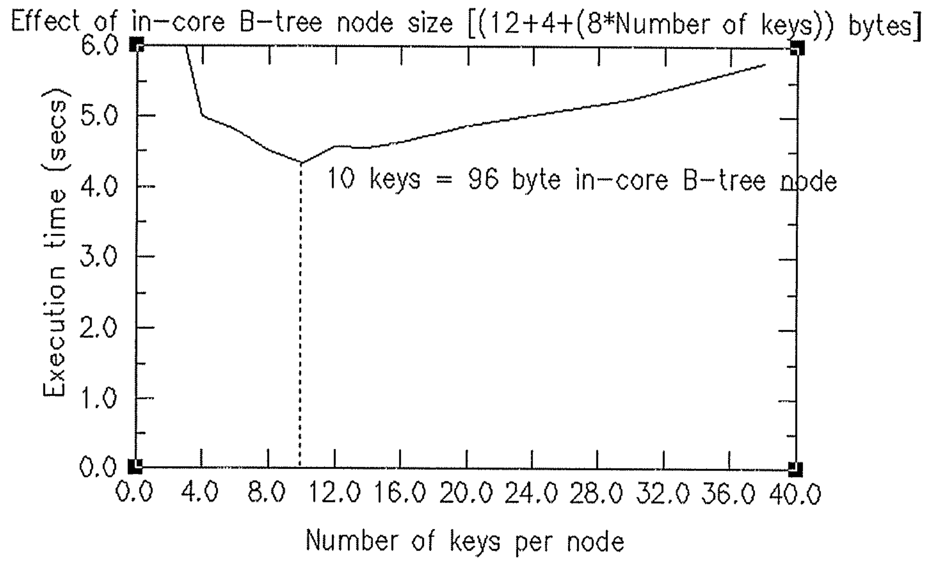

Figure 16. B-tree node size.

the average sized node fits in a cache block, and not the largest in-core B-tree node. This result does not hold for out-of-core B-trees due to the enormous penalty of accessing an additional disk page.