Making Pointer-Based Data Structures Cache Conscious

To narrow the widening gap between processor and memory performance, the authors propose improving the cache locality of pointer-manipulating programs and bolstering performance by careful placement of structure elements.

Trishul M. Chilimbi
Microsoft Research

Mark D. Hill
University of Wisconsin-Madison

James R. Larus
Microsoft Research

Rapid increases in processor speed and far slower increases in memory speed have led to memory access times that far exceed the cost of simple arithmetic operations. The ubiquitous hardware solution to this problem is memory caches (see "Cache Memory" sidebar), which exploit program locality to reduce the average latency. Other techniques—such as prefetching, multithreading, nonblocking caches, dynamic instruction scheduling, and speculative execution—use complex hardware and software to further reduce or hide the high cost of memory accesses.

Despite these mechanisms, the processor-memory gap currently requires a hierarchy of two or more caches between the processor and memory. The wide range of costs of finding data in this hierarchy undercuts the fundamental RAM model assumption that all memory accesses have unit cost, which is the model most programmers use to understand and design data structures and algorithms. This divergence between theory and practice underlies many performance problems.

Programming languages have also evolved. Early languages such as Fortran and Algol stored data in array structures. Later languages, such as Simula, Pascal, C, and C++ supported pointers, which facilitated writing programs such as databases and operating systems that make extensive use of pointer structures. With their dynamic nature and reliance on heap-allocated storage, pointer structures have fewer regular access patterns than do arrays. Not surprisingly, techniques that reduce or tolerate array structure access latency are not as effective when used in pointer-manipulating applications.[1] Many of these techniques are fundamentally limited by their focus on the manifestation of the problem—memory latency—rather than on its cause—poor reference locality.

In general, changing a program's data access pattern or data organization and layout can improve reference locality. Compilers can analyze array accesses and perform transformations that reorder these accesses without affecting a program's result. Compilers use two array-structure properties to reorder the accesses: uniform, random access to array elements, and a number-theoretic basis for statically analyzing data dependencies.

Pointer structures, however, share neither property. For example, a key search in a tree structure must start at the root and follow tree-node pointers to the appropriate leaf of the tree. Generally, it is impossible to reorder these accesses. In addition, although there has been much progress in improving pointer-analysis techniques, they can rarely guarantee that reordering pointer accesses will not affect a program's result.

Pointer structures consist of separate, independently allocated pieces, and they possess the powerful property of location transparency. Because of this location transparency, pointer structures can place elements in a compound data structure in different memory and cache locations without changing a program's semantics. The careful placement of structure elements provides the essential mechanism for improving the cache locality of pointer-manipulating programs and, consequently, their performance.

0018-9162/00/$10.00 © 2000 IEEE

Cache Memory

Cache memory is constrained to be small to ensure high-speed access. Thus, cache capacity is much smaller than main-memory capacity. To amortize the high cost of accessing main memory, the system transfers data in cache blocks, or lines; these units encompass multiple words, typically 16 to 128 bytes. To limit the blocks that a cache simultaneously searches, the cache typically constrains block placement to one, two, or four cache locations. A cache's associativity is the number of locations where it can place a block. Figure A shows these design constraints.

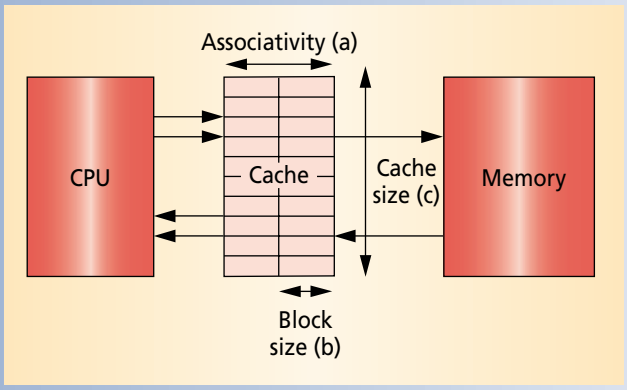

Figure A. In a memory cache, important cache parameters include block size, which determines the transfer unit between memory and cache; cache capacity which determines how much data the cache holds; and cache associativity, which constrains the number of distinct locations where the cache can place a block.

A program's miss rate—the fraction of the total number of references that miss in the cache and thus must access main memory—often characterizes its cache performance. Higher miss rates indicate poorer cache performance. The following formula gives the average memory-access time for a machine architecture with a cache:

Access time = cache hit time + (cache miss rate × cache miss penalty)

Because the underlying hardware determines cache hit time and miss penalty, reducing the cache miss rate provides the primary opportunity for software writers to improve a program's memory system performance.

It is sometimes useful to characterize cache misses as

- *compulsory misses*, which occur when a data item is first loaded in the cache,
- *capacity misses*, which would generate hits in a larger cache, and
- *conflict misses*, which result from limited cache associativity and arise from different blocks mapping to the same position in the cache.

DESIGNING CACHE-CONSCIOUS DATA STRUCTURES

Figure 1 shows different approaches for improving cache performance. The shaded units indicate data items accessed at the same time. Figure 1a shows the implicit prefetching achieved by packing cache blocks with contemporaneously accessed data. Figure 1b shows why this packing reduces compulsory and capacity misses. Figure 1c shows why mapping concurrently accessed structure elements (which do not fit in a single cache block) to nonconflicting cache blocks reduces conflict misses.

Programmers can combine these three general data placement design principles—clustering, coloring, and compression—to produce cache-conscious data structures.

Clustering

Clustering packs data structure elements that the program is likely to access at the same time in a cache block, thus improving spatial and temporal locality and providing implicit prefetching.

For example, an effective way to cluster a tree is to pack subtree regions into a cache block. For a series of random binary 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 greater than 2 for $k > 3$. In the case of a depth-first clustering scheme, where the k nodes in a block form a single parent-child-grandchild chain, the expected number of accesses to the block per tree search is bounded by 2.

This analysis assumes a random access pattern. For specific access patterns, such as a depth-first search, other clustering schemes may be better. In addition, tree modifications can destroy locality. However, our studies indicate that for trees that change infrequently, subtree clustering is more efficient than allocation-order clustering, which places contemporaneously allocated tree nodes in the same cache block.

Coloring

Caches have finite associativity—only a limited number of concurrently accessed data elements can map to the same cache block without causing conflict misses. Coloring places elements in memory such that elements accessed at the same time map to nonconflicting cache regions. Figure 2 shows a two-color scheme, which can extend easily to multiple colors, for a two-way, set-associative cache. This coloring partitions a cache with C cache sets (each set containing a, where a is *associativity* blocks) into two regions: p sets, and $C - p$ sets. The coloring uniquely maps frequently accessed structure elements to the first cache region so that they do not conflict with each other, whereas it maps the remaining elements to the other region.

Such a mapping avoids conflicts among heavily accessed data structure elements and prevents infrequently accessed elements from replacing them. In addition, if the gaps in the virtual address space that implement coloring are a multiple of the virtual memory page size, this scheme does not waste any physical memory.

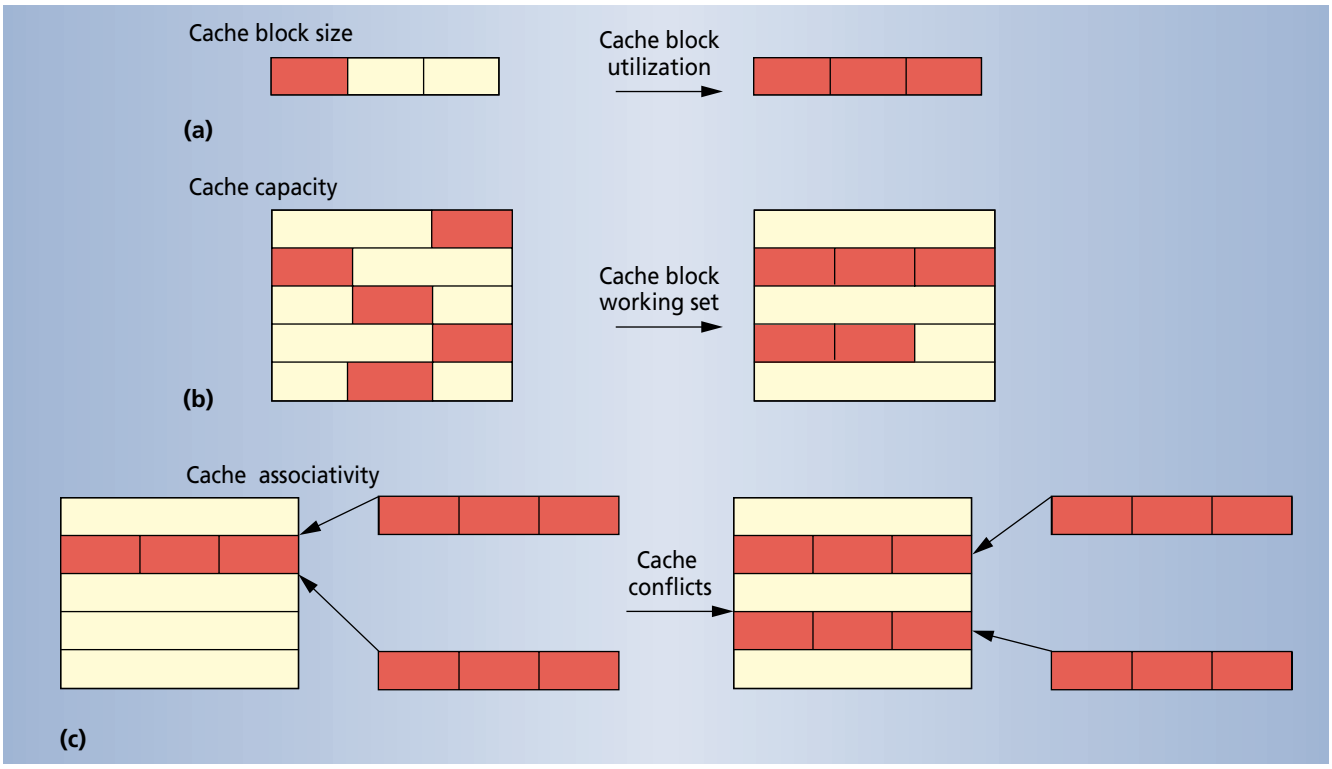

Figure 1. Several approaches can improve cache performance. (a) Packing cache blocks with contemporaneously accessed data improves cache block utilization. (b) Applying a similar technique across multiple cache blocks reduces the cache block working set. (c) Mapping contemporaneously accessed data to different cache locations reduces cache conflicts.

Compression

Compressing data structure elements lets more elements cluster in a cache block, increasing cache block utilization and shrinking a structure's memory footprint, thereby reducing capacity and conflict misses. Compression typically requires additional processor operations to decode compressed information; however, with high memory access costs, this computation may be cheaper than additional memory references. Structure compression methods include data encoding techniques, such as key compression,[2] and structure encoding techniques, such as

- Pointer elimination, which replaces pointers with computed offsets. The implicit heap data structure, a classic example, stores a node's children at known offsets in an array.
- Hot and cold structure splitting, which works because most data-structure searches examine only a portion of individual elements until they find a match. Structure splitting separates heavily accessed (hot) portions of data structure elements from rarely accessed (cold) portions. Although this technique increases the total size of the data structure slightly, it can significantly reduce the size of the hot working set.

CACHE-CONSCIOUS DATA PLACEMENT STRATEGIES

Although cache-conscious pointer structure design offers important performance benefits, this approach is difficult for average programmers because it requires

- a complete understanding of an application's code and data structures,
- knowledge of the underlying cache architecture—something many programmers are unfamiliar with—and
- significant rewriting of an application's code.

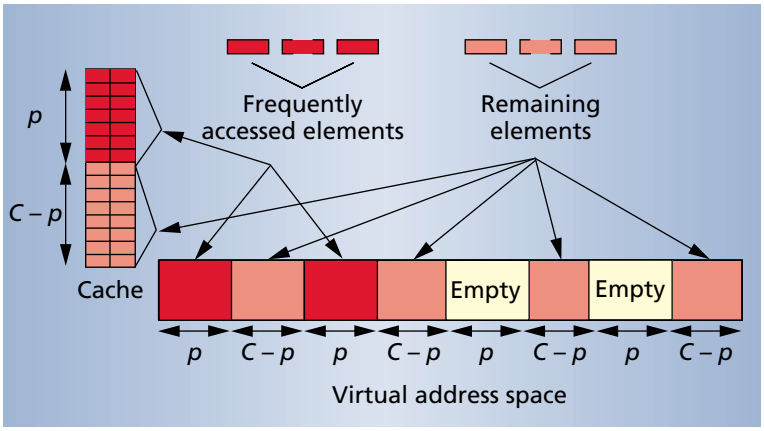

Figure 2. Coloring data structure elements reduces cache conflicts. The two-color scheme shown divides a cache with C cache sets into two regions: p *sets and* C – p *sets. This coloring uniquely maps frequently accessed elements to a portion of the cache so that infrequently accessed elements do not displace them. The coloring accomplishes this by inserting gaps in the virtual address space that remain empty.*

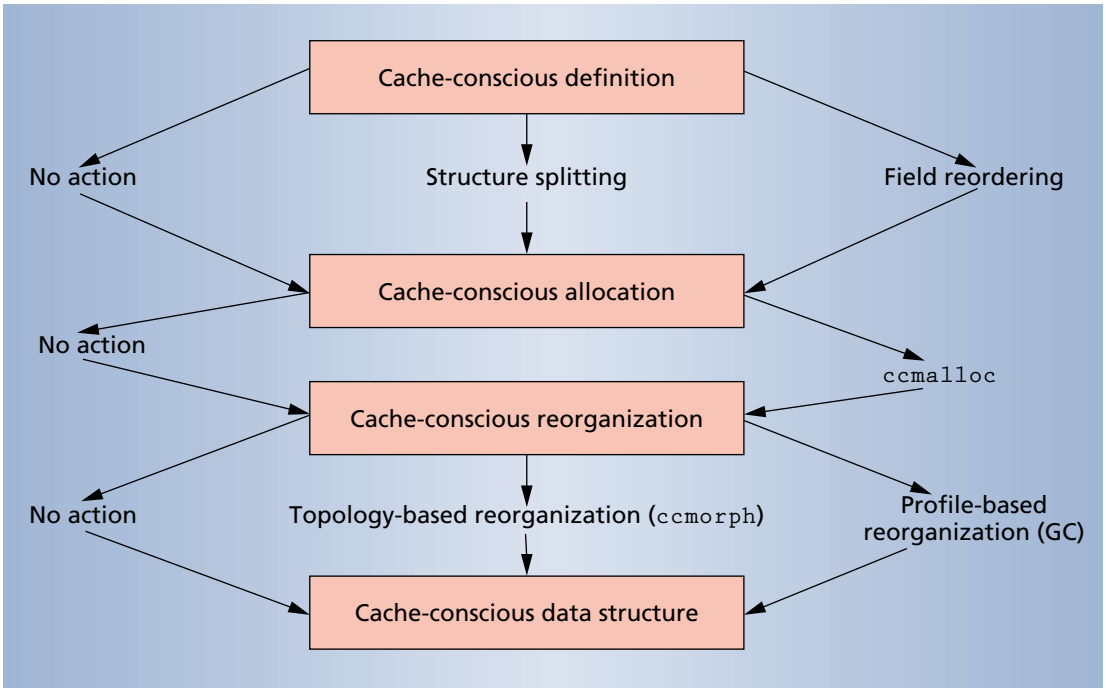

Figure 3. Software tool writers can combine different strategies for producing cache-conscious data structures in a variety of ways to produce a cache-conscious data structure. The different strategies are orthogonal and complement each other.

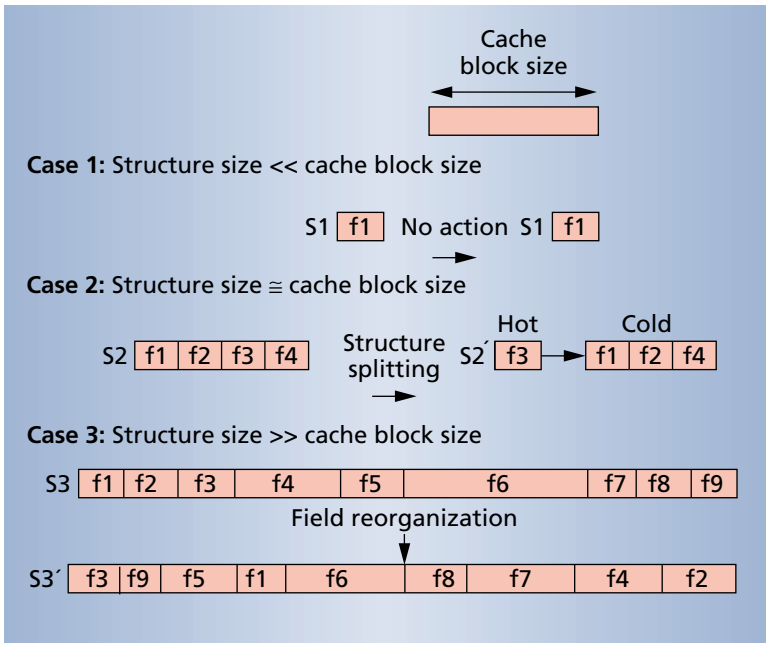

Figure 4. Cache-conscious structure definition depends on the size of structure elements. In case 1, the structure has a single field and is much smaller than the cache block size so no action is necessary at definition time. In case 2, where the structure element size is comparable with the cache block size, splitting allows a clustering scheme to pack multiple hot structure instances in the same cache block. Finally, in case 3, where the structure size is much larger than the cache block size, field reordering may improve cache block utilization.

To address these problems, we designed several automatic and semiautomatic strategies that produce cache-conscious pointer structures. These strategies make cache-conscious data structures available to average programmers, just as compilers made high-performance programming accessible to those who could not use assembly language. The strategies apply clustering, coloring, and compression to make existing pointer structures cache conscious.

Our strategies use three methods to construct cache-conscious pointer structures:

- *Changing a structure's definition.* Changing the definition permits clustering of fields accessed contemporaneously. Splitting structures into hot and cold portions based on program accesses permits packing more hot instances, which are accessed together, in the same cache block. Both techniques increase cache block utilization.
- *Modifying the allocation policy for structure elements.* Cache-conscious allocation attempts to collocate contemporaneously accessed data elements in the same physical cache block at allocation time. This improves cache performance by increasing cache block utilization.
- *Reorganizing the structure layout.* Cache-conscious reorganization transforms the pointer structure layout into a linear memory layout, sequenced according to the expected data access pattern, and maps structure elements to reduce

cache conflicts. The reorganizer obtains the expected access pattern from program profiles. For certain pointer structures such as trees, the reorganizer can glean access information from data structure topology.

Figure 3 shows how to combine various strategies to produce cache-conscious data structures.

STRUCTURE DEFINITION

Structure splitting and field reordering, two cache-conscious definition techniques, can improve a program's cache behavior. Figure 4 shows the relationship of cache-conscious definition techniques and the size of structure elements. The three possibilities depend on the size of structure elements relative to the cache block size.

Structure splitting

Many Java objects are comparable in size to a cache block (case 2 in Figure 4).[3] Because Java is a type-safe language, we can automate class (structure) splitting. To do so, we first identify class member fields as hot or cold. Although we can use static analysis to classify some member fields, profiling a program to determine field access frequency seems to be a simpler, more general approach. A compiler extracts cold fields from the class and places them in a new object, to which the original object references indirectly. Accesses to cold fields require an extra indirection to the new class, whereas accesses to hot fields remain unchanged.

Splitting's overhead includes the space cost of an additional reference from the hot portion to the cold portion, code bloat, more objects in memory, and an extra indirection for accesses to cold fields. Our splitting algorithm takes these factors into account and is designed to reduce these costs.[3] In addition, our garbage collection scheme for cache-conscious object collocation aggressively exploits the advantage that smaller hot-class instances provide by packing more hot instances in the same cache block.

For five medium-sized Java benchmarks, including Javac (a bytecode compiler) and Javadoc (a document generator), class splitting combined with our garbage collection scheme for cache-conscious object collocation reduced L2 cache miss rates by 29 to 43 percent, with class splitting accounting for 26 to 62 percent of this reduction. Execution time improved by 18 to 28 percent; class splitting contributed 22 to 66 percent of this improvement.[3] To run these experiments, we used a single processor of a 167-MHz Sun Ultraserver, E5000 system, with an optimizing Java compiler that generates native Sparc assembly code. For languages such as C and C++, which do not permit automatic structure splitting, the algorithm's splitting recommendations can provide programmer feedback.

Field reordering

Many legacy applications were designed when machines lacked multiple levels of cache and memory-access times were more uniform. In particular, commercial C applications often manipulate large structures. In this case, structure splitting will likely produce hot elements larger than a cache block, which makes splitting ineffective. Reordering structure fields to place those with high temporal affinity in the same cache block can improve cache-block utilization. Typically, programmers group fields in large structures conceptually, which may not correspond with their temporal access pattern. Unfortunately, the logical order can cause structure references to interact poorly with a program's data-access pattern, resulting in unnecessary cache misses. Compilers for many languages are constrained to follow the programmer-supplied field order and, therefore, cannot correct this problem.

The logical order can cause structure references to interact poorly with a program's data-access pattern, resulting in unnecessary cache misses.

To investigate field-reordering benefits, we implemented an algorithm for recommending reordering of structure fields in C programs. This algorithm correlates static information about the source location of structure-field accesses with dynamic information about the temporal ordering of accesses and their execution frequency. The algorithm uses the data to construct a field-affinity graph for each structure and then processes these graphs to produce field-order recommendations. Measurements obtained from a four-processor 400-MHz Pentium II Xeon system with a 1-Mbyte L2 cache, 4 Gbytes of memory, and 200 7,200-rpm Clariion Fibre Channel disk drives indicate that reordering fields in five active structures improves the performance of Microsoft SQL Server 7.0, a large, highly tuned commercial application, by 2 to 3 percent on the Transaction Processing Council C benchmark.[3]

STRUCTURE ALLOCATION

Programmers typically allocate a data structure's elements with little concern for memory hierarchy. Often the resulting layout may interact poorly with the program's data access patterns, causing unnecessary cache misses and reducing performance. To address this problem, cache-conscious allocation collocates contemporaneously accessed data elements in the same cache block. Because a program invokes a heap allocator many times, a cache-conscious allocator must use techniques that incur low overhead. Further, a heap allocator has an inherently local view of a structure. For these reasons, our cache-conscious heap allocator (`ccmalloc`) only performs local clustering. `Ccmalloc` is safe; incorrect usage affects only program performance, not correctness.

A memory allocator similar to `malloc`, `ccmalloc` takes an additional parameter that points to an existing

Figure 5. Code from the Olden health benchmark.

```
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;
}
```

data structure element that the program is likely to access contemporaneously with the element to be allocated, such as the tree node's parent. The allocator attempts to locate the new data item in the same cache block as the existing item. This code from the Olden health benchmark illustrates this approach in Figure 5. Our experience with `ccmalloc` indicates that even a programmer unfamiliar with an application can often select a suitable parameter and obtain good results by examining the code surrounding the allocation statement.

In a memory hierarchy, different cache block sizes mean that the allocator can collocate data in different ways. The `ccmalloc` allocator focuses on L2 cache blocks. In the Sun UltraSparc 1 we used in this study, L1 cache blocks are effectively only 16 bytes (L2 blocks are 64 bytes), which severely limits the number of objects that fit in a block. Moreover, the bookkeeping overhead in the allocator is inversely proportional to the size of a cache block, so larger blocks are more likely to be successful and incur less overhead. On a system with a larger L1 cache block, adopting a hierarchical approach may be advantageous, with collocation first attempted in the same L1 cache block. If this fails, the allocator could attempt to make the subsequent collocation in the same L2 cache block.

Cache-conscious heap allocation with `ccmalloc` resulted in a speedup of 27 percent for VIS, a 160,000-line system that uses binary decision diagrams (directed acyclic graphs) to formally verify finite state systems.[4] We obtained these results on a 167-MHz Sun Ultraserver E5000. Significantly few changes to the program (that is, fewer than 300 code lines) produced these large performance improvements, indicating that cache-conscious data placement can even improve the performance of graphlike data structures in which data elements have multiple parents.

STRUCTURE REORGANIZATION

Reorganizing a structure's memory layout to correspond with its access pattern is a complementary approach to cache-conscious allocation. Although you perform cache-conscious allocation just once when you create a data element, you can use cache-conscious reorganization as often as required. Successful memory layout reorganization of general graphlike structures requires a detailed profile of a program's data access patterns.[5,6] However, trees are an important class of structures possessing topological properties that permit cache-conscious data reorganization without profiling.

A transparent/semantics-preserving cache-conscious tree reorganizer, `ccmorph`, applies the clustering and coloring techniques described earlier. This reorganizer is appropriate for a "read-mostly" data structure that a program builds early in a computation and subsequently references heavily. With this approach, a program doesn't need to change either the construction or the consumption code because the reorganizer can reorganize the structure between the two phases. Moreover, if the structure changes slowly, the reorganizer can invoke `ccmorph` periodically.

Languages that support garbage collection offer a more attractive alternative. Copying garbage collectors, which support automatic memory management, determine when dynamically allocated storage becomes unreachable; the program then automatically recycles that memory by traversing the heap and copying live data to a separate memory region.

The copying phase frees up all memory in the traversed space for reuse. The copying phase of garbage collection offers an invaluable opportunity to reorganize a program's data layout to improve cache performance. However, such a scheme relies on the ability to transparently relocate heap data. In addition, it requires a differentiation between pointers and nonpointer data. Hence, you cannot implement a garbage collection scheme as described for low-level languages, such as C or C++, which support arbitrary pointer-manipulation operations and preclude transparent data movement. Chi-Keung Luk and Todd C. Mowry have proposed new hardware mechanisms that remove this obstacle.[7] Object-oriented languages such as Java and Cecil and functional languages such as ML and Lisp permit copying garbage collection. For these languages, a copying garbage collector can reorganize data and produce a cache-conscious structure layout.

Topology-based structure reorganization

In languages such as C that support unrestricted pointers, analytical techniques cannot precisely iden-

tify all pointers to a structure element. Without this knowledge, a system cannot move or reorder data structures without an application's cooperation, as it can in a language designed for garbage collection.[6] However, if a programmer guarantees the safety of the transformation, ccmorph applies clustering and coloring techniques to improve structure access locality by transparently reorganizing the tree data structure.

The ccmorph reorganizer operates on treelike structures that have homogeneous elements and do not have external pointers to the structure's middle (or on any data structure that can be decomposed into components satisfying this property). However, ccmorph supports a liberal definition of a tree in which elements can contain a parent or predecessor pointer. A programmer supplies ccmorph with a pointer to a data structure's root, a function to traverse the structure (*next_node*), and cache parameters. For example, the following code reorganizes the quadtree data structure in the Olden *perimeter* benchmark, with the programmer supplying the *next_node* function:

```
main()
{
  ...
  root = maketree(4096, ..., ...);
  ccmorph(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);
 }
}
```

The ccmorph reorganizer copies a structure into a contiguous block of memory (or a number of contiguous blocks, in the case of large structures). As Figure 6 shows, in the process, it partitions a tree-like structure into subtrees laid out linearly. The reorganizer colors this structure to map the first *p* elements traversed to a unique portion of the cache, determined by the *Color_const* parameter, that will not conflict with other structure elements. In addition, ccmorph determines the values of *p* and the size of subtrees from the cache parameters and the structure element size. The ccmorph reorganizer ensures that the gaps in the virtual address space that implement coloring correspond with multiples of the virtual-memory page size.

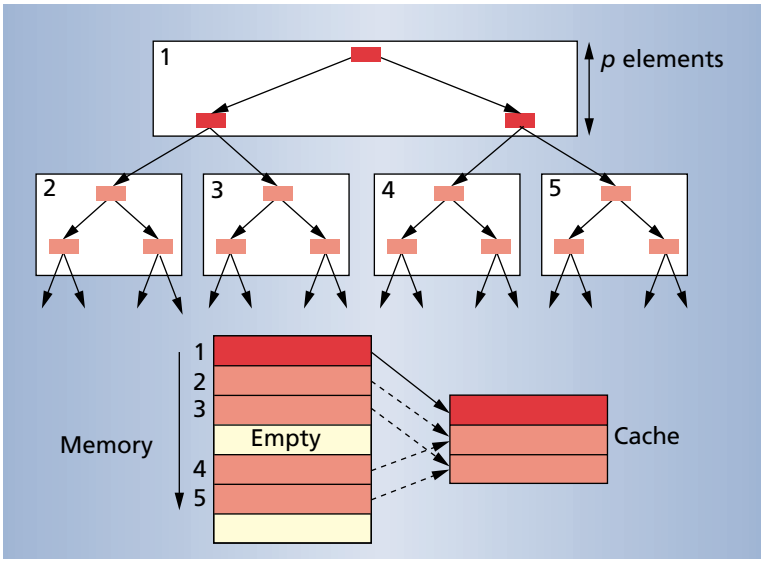

Figure 6. Cache-conscious tree reorganization applies subtree clustering to the entire tree. The reorganizer allocates the top levels—frequently accessed elements—of the tree map to map to a portion of the cache where lower level tree nodes—infrequently accessed elements—cannot displace them.

Ccmorph was used to optimize the performance of Radiance, a 60,000-line program for modeling the distribution of visible radiation in an illuminated space. Radiance's primary data structure is an octree that represents the scene it is modeling. Cache-conscious clustering and coloring of the octree produced a speedup of 42 percent, including the overhead of restructuring the octree, on a 167-MHz Sun Ultraserver E5000 system.[4]

Profile-based structure reorganization

A cache-conscious data layout places objects with high temporal affinity near one another so they can reside in the same cache block. In this approach, a program's accesses are profiled. The profile-based reorganizer (garbage collector) uses the profiling data it gathers during an execution to optimize that execution rather than a subsequent one. We rely on a property of object-oriented programs—most objects are small—to perform low-overhead real-time data profiling.[3,6] The garbage collector uses this profile to construct an object affinity graph in which weighted

edges encode the temporal affinity between objects or nodes. A new garbage collection copying algorithm makes a depth-first traversal of the affinity graph to produce cache-conscious data layouts while copying objects. The technique is automatic and requires no programmer intervention.

Experimental results for several object-oriented programs show that this cache-conscious data placement technique reduces cache miss rates by 16 to 42 percent and improves program performance by 10 to 37 percent, including real-time data-profiling overhead.[3,6] Further, we used one processor of a 167-MHz Sun Ultraserver E5000 system to compare our cache-conscious copying scheme with the Wilson-Lam-Moher algorithm.[8] This earlier algorithm attempted to improve a program's virtual memory (page) locality by changing the traversal algorithm. The results showed that our cache-conscious object-layout technique reduces cache miss rates by 14 to 41 percent and improves program performance by 8 to 31 percent compared with the Wilson-Lam-Moher technique. This finding indicates that page-level improvements are not necessarily effective at the cache level.[6]

Considering past trends and future technology, it seems clear that the processor-memory performance gap will continue to increase and software will continue to grow larger and more complex. Although cache-conscious algorithms and data structures are the first and perhaps best place to attack this performance problem, the complexity of software design and an increasing tendency to build large software systems by assembling smaller components does not favor a focused, integrated approach. We propose another, more incremental approach of cache-conscious data layout, which uses techniques such as clustering, coloring, and compression to enhance data locality by placing structure elements more carefully in the cache. ✸

Acknowledgments

This research was conducted as partial fulfillment of Trishul M. Chilimbi's doctoral thesis, "Cache-Conscious Data Structures—Design and Implementation," at the University of Wisconsin-Madison. This study was supported in part by the National Science Foundation (MIPS-9625558, CCR-9357779, EIA-9971256, and CDA-9623632), Microsoft Corporation, and Sun Microsystems. We thank Craig Chambers and Dave Grove for the Vortex compiler infrastructure; members of the Wisconsin Wind tunnel Project; Bob Davidson and members of Microsoft Research's Advanced Development Tools Group; and the Semantics-Based Tools Group at Microsoft Research. Thomas Ball, Ras Bodik, Milo Martin, and Dan Sorin provided constructive comments.

References

1. S.E. Perl and R.L. Sites, "Studies of Windows NT Performance Using Dynamic Execution Traces," *Proc. 2nd Usenix Symp. Operating Systems Design and Implementation*, ACM Press, New York, 1996, pp. 169-183.
2. D. Comer, "The Ubiquitous B-Tree," *ACM Computing Surveys*, June 1979, pp. 121-137.
3. T.M. Chilimbi, B. Davidson, and J.R. Larus, "Cache-Conscious Structure Definition," *Proc. SIGPLAN 99, Conf. Programming Language Design and Implementation*, ACM Press, New York, 1999, pp. 13-26.
4. T.M. Chilimbi, M.D. Hill, and J.R. Larus, "Cache-Conscious Structure Layout," *Proc. SIGPLAN 99, Conf. Programming Language Design and Implementation*, ACM Press, New York, 1999, pp. 1-12.
5. B. Calder et al., "Cache-Conscious Data Placement," *Proc. 8th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1998, pp. 139-149.
6. T.M. Chilimbi and J.R. Larus, "Using Generational Garbage Collection to Implement Cache-Conscious Data Placement," *Proc. Int'l Symp. Memory Management*, ACM Press, New York, 1998, pp. 37-48.
7. C.-K. Luk and T.C. Mowry, "Memory Forwarding: Enabling Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation," *Proc. 26th Ann. Int'l Symp. Computer Architecture*, ACM Press, New York, 1999, pp. 88-99.
8. P.R. Wilson, M.S. Lam, and T.G. Moher, "Effective 'Static-Graph' Reorganization to Improve Locality in Garbage-Collected Systems," *SIGPLAN Notices*, June 1991, pp. 177-191.

Trishul M. Chilimbi *is a researcher at Microsoft. His research interests include programming languages, compilers, computer architectures, and parallel and distributed systems. He received a PhD in computer science from the University of Wisconsin-Madison. Contact him at trishulc@microsoft.com.*

Mark D. Hill *is a professor in the Computer Sciences Department and the Electrical and Computer Engineering Department at the University of Wisconsin-Madison. His research interests include multiprocessor and uniprocessor memory systems. He received a PhD in computer science from the University of California, Berkeley. Contact him at markhill@cs.wisc.edu.*

James R. Larus *is a senior researcher at Microsoft. His research interests include programming languages, compilers, parallel computation, and software tools. He received a PhD in computer science from the University of California, Berkeley. Contact him at larus@microsoft.com.*