Making Pointer-Based Data Structures Cache Conscious

Trishul M. Chilimbi
Microsoft Research
One Microsoft Way
Redmond, WA 98052
trishulc@microsoft.com

Mark D. Hill
Computer Sciences Dept.
University of Wisconsin
1210 W. Dayton St.
Madison, WI 53706
markhill@cs.wisc.edu

James R. Larus
Microsoft Research
One Microsoft Way
Redmond, WA 98052
larus@microsoft.com

Abstract

Processor and memory technology trends portend a continual increase in the relative cost of accessing main memory. Machine designers have tried to mitigate the effect of this trend through a hierarchy of caches and a variety of other hardware and software techniques. These techniques, unfortunately, have only been partially successful for pointer-manipulating programs.

This paper explores a complementary approach of enlisting programmers and tool writers in the task of improving the cache locality of accesses to pointer-based data structures. Throughout, we exploit the location transparency of pointer-based data structures that allow changes to the memory (and cache) layout of nodes, records, fields, etc. We discuss how programmers can manually improve cache performance with techniques, such as clustering, compression, and coloring. We then explore how to lessen a programmer's burden with the help of semi-automatic and automatic tools for changing structure layout to improve cache performance. Techniques include reordering fields in a structure definition, carefully placing nodes in memory at allocation time, and dynamically reorganizing extant data structures.

Keywords: cache; memory system; pointer data structures; data layout; data organization; data restructuring; field reordering; memory allocation; structure splitting; performance speedup; performance tools.

1 Introduction

Processor and memory technology trends portend a continual increase in the relative cost of accessing main memory, so that today *a memory access can be hundreds of times more costly than an arithmetic operation*. Memory caches are the ubiquitous hardware solution to this problem (see Sidebar). These are small, fast memories that store recently accessed data items and attempt to intercept and satisfy data requests without accessing main memory. Before the 1990s, a single unified cache sufficed. Today the increasing performance gap (now two orders of magnitude) makes necessary a cache hierarchy where instruction fetches access an instruction cache (ignored in this paper), data accesses use a data cache, and misses from both are serviced by one or two levels of even larger unified caches.

In addition to caches, a variety of hardware and software techniques—such as prefetching, multithreading, non-blocking caches, dynamic instruction scheduling, and speculative execution—have been developed and implemented to reduce or tolerate memory latency. Despite these techniques, which require complex hardware or software, many programs' execution time is dominated by the latency of memory references. Moreover, high and variable memory access costs undercut the fundamental random-access memory (RAM) model (all memory accesses have unit cost) that most programmers use to understand and design data structures and algorithms. This can cause unexpected behavior; for example, algorithms with a larger number of operations (but fewer memory references) may outperform alternative algorithms with fewer operations.

From a software perspective, programming languages used to write programs have also evolved. Early languages such as Fortran and Algol, used mainly for scientific applications, did not support pointers. Applications written in these languages store their data in array structures. Subsequent languages such as Simula, Pascal, C, and C++ supported pointers. Many applications written in these languages, such as

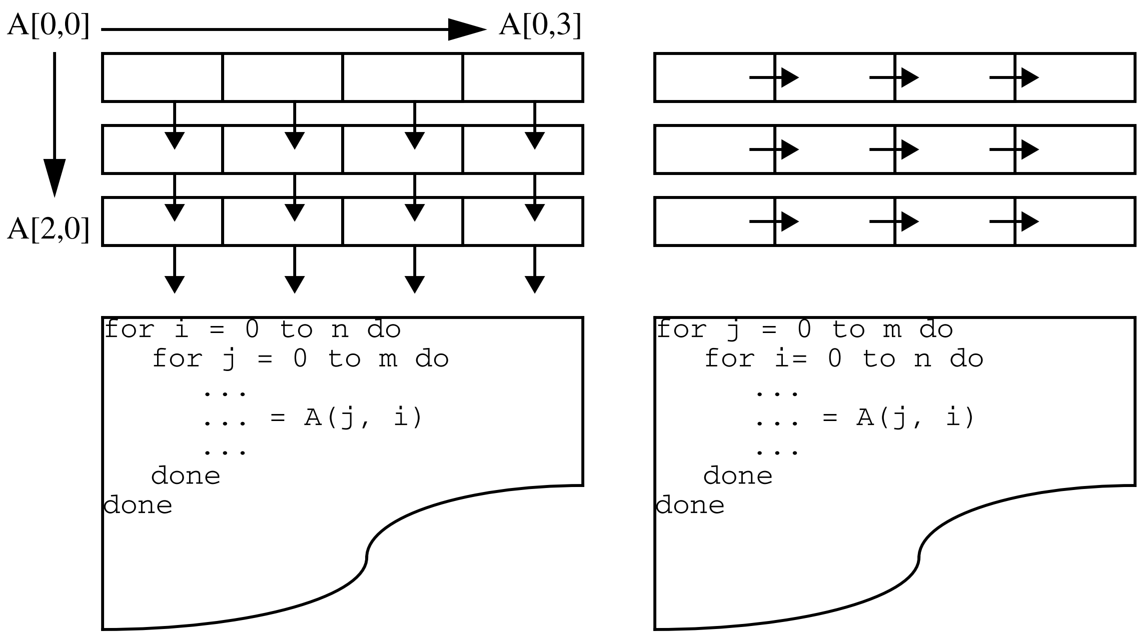

Figure 1. Improving software locality through program transformation. In the worst case, the code snippet on the left will suffer a cache miss on every access, whereas that on the right will incur a cache miss only on every fourth access.

databases and operating systems, make extensive use of pointer structures to store data. Due to their dynamic nature and reliance on heap-allocated storage, pointer structures tend to have less regular access patterns than array structures.

We call applications that make extensive use of pointer structures *pointer-manipulating programs*. Not surprisingly, techniques for reducing and tolerating latency that were developed primarily for applications that manipulate data stored in array structures are not as effective for pointer-manipulating programs [10]. In addition, many techniques are fundamentally limited by their focus on the manifestation of the problem (memory latency), rather than its cause (poor reference locality).

In general, reference locality can be improved either by changing a program's data access pattern or its data organization and layout. The first approach has been successfully applied to improve the cache locality of scientific programs that manipulate dense matrices [12, 2, 8]. Figure 1 illustrates this with an example. If it is possible to interchange the loops as shown in the code snippet on the right, the resultant data reference pattern will step through all array elements in a cache block before accessing the next block, reducing the number of potential cache misses by a factor of four. Two properties of array structures—uniform, random access to elements, and a number-theoretic basis for statically analyzing data dependencies—allow compilers to analyze array accesses and perform transformations that reorder accesses without affecting a program's result

Unfortunately, pointer structures share neither property. Consider, for example, a tree structure. A key search on this tree structure has to start at the root of the tree and follow tree node pointers to the appropriate leaf. Reordering these accesses is, in general, impossible. In addition, although much progress has been made in pointer-analysis techniques, they are still not strong enough to guarantee that reordering pointer accesses will not affect a program's result. Pointer structures are however composed of separated, independently allocated pieces and possess an extremely powerful property of *location transparency*: elements in a compound data structure can be placed at different memory (and cache) locations without changing a program's semantics. The thesis of this work is that careful placement of structure elements provides the essential mechanism to improve the cache locality of pointer-manipulating programs, and consequently, their performance.

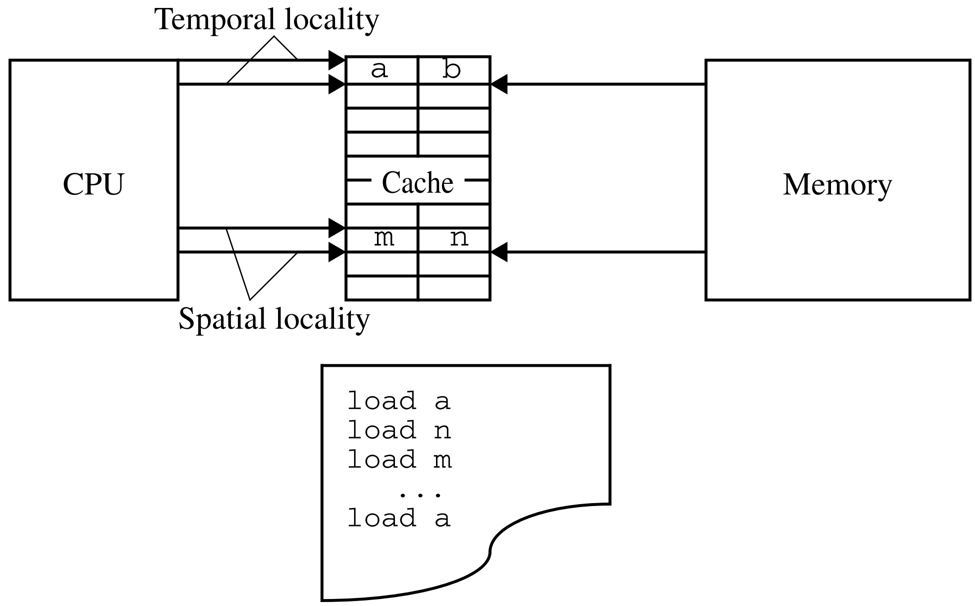

Figure 2. Exploiting locality with a cache. The second access to *a* could be satisfied without a memory transfer due to its temporal proximity to a prior access to *a*. The load of *m* will also not require a memory access since *m* is in the same cache block as *n*, which was brought into the cache by the previous access.

This paper seeks to make two contributions in bringing our recent research results [3,4,5,6] to a broad professional audience. First, we explain how programmers can manually improve cache performance with techniques, such as clustering, compression, and coloring (Section 2). Second, we explore how to lessen a programmer's burden via semi-automatic and automatic tools that improve cache performance by reordering fields when structures are defined, carefully placing nodes in memory at allocation time, and dynamically reorganizing extant data structures (Section 3).

SIDEBAR: Cache Memory

Caches are small, fast memories that store recently accessed data items and attempt to intercept and satisfy data requests without accessing main memory [1]. Caches can improve performance by exploiting data reference locality (see Figure 2). There are two types of data locality—*temporal* and *spatial*. A data item exhibits temporal locality if it is repeatedly accessed within a short period of time. Spatial locality implies that data items stored in adjacent memory locations are likely to be accessed contemporaneously. Caches exploit temporal locality by storing recently accessed data. Caches transfer data from main memory in contiguous blocks that encompass multiple words and, consequently, benefit from spatial locality.

Cache memory is constrained to be small to ensure high-speed access, and hence cache capacity is much smaller than main memory capacity. To amortize the high cost of accessing main memory, data is transferred in units called cache *blocks* (or *lines*) that encompass multiple words (typically 16-128 bytes). To limit the blocks simultaneously searched on a cache access, block placement in the cache is typically constrained to 1, 2, or 4 locations. The number of locations where a block can be placed is a cache's *associativity*. Figure 3 illustrates these design constraints.

A program's cache performance is often characterized by its *miss rate*. This is the fraction of the total number of references that miss in the cache and need to access main memory. Higher miss rates indicate poorer cache performance. The average memory-access time for a machine architecture with a cache is given by

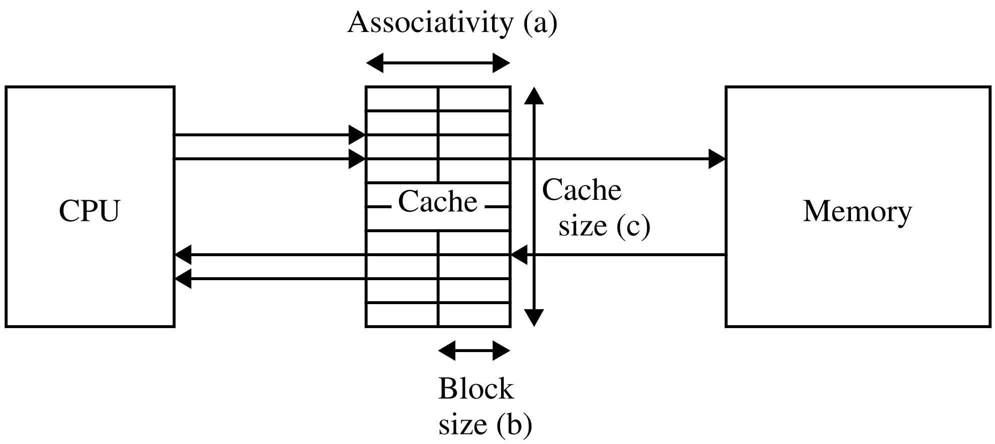

Figure 3. Memory cache. Important cache parameters include block size, which determines the transfer unit between memory and cache, cache capacity, and cache associativity, which constrains the number of distinct cache locations that a block can be placed.

$$Access\ Time = Cache\ Hit\ Time + (Cache\ Miss\ Rate) \times (Cache\ Miss\ Penalty)$$

Since cache hit time and miss penalty are determined by the underlying hardware, reducing the cache miss rate provides the only opportunity to improve a program's memory system performance.

It is sometimes useful to characterize cache misses as *compulsory* misses, *capacity* misses, and *conflict* misses. Compulsory misses are incurred when a data item is first loaded in the cache. A miss is a capacity miss if it would hit in a cache of larger size. Finally, a conflict miss is a result of limited cache associativity and arises from different blocks mapping to the same position in the cache.

Reference

[1] John L. Hennessy and David A. Patterson. "Computer Architecture: A Quantitative Approach." *Morgan Kaufmann Publishers Inc.*, San Mateo, CA, 1990.

2 Designing Cache-Conscious Data Structures

Figure 4 illustrates different approaches to improving cache performance. The shaded units indicate contemporaneously accessed data items. Figure 4a illustrates the implicit prefetching achieved by packing cache blocks with contemporaneously accessed data. More globally, Figure 4b illustrates why this packing reduces both compulsory and capacity misses. Finally, Figure 4c illustrates why mapping concurrently accessed structure elements (which do not fit in a single cache block) to non-conflicting cache blocks reduces conflict misses.

This section discusses three general data placement design principles that can be combined in a wide variety of ways to produce cache-conscious data structures. *Clustering* attempts to pack data structure elements likely to be accessed contemporaneously into a cache block. This increases cache block utilization and reduces the cache block working set (see Figure 4). *Coloring* segregates heavily and infrequently accessed elements in non-conflicting cache regions. This reduces cache conflicts. *Compression* reduces structure size or separates the active portion of structure elements. This increases the benefits that arise from applying clustering or coloring.

2.1 Clustering

Clustering attempts to pack in a cache block, data structure elements likely to be accessed contem-

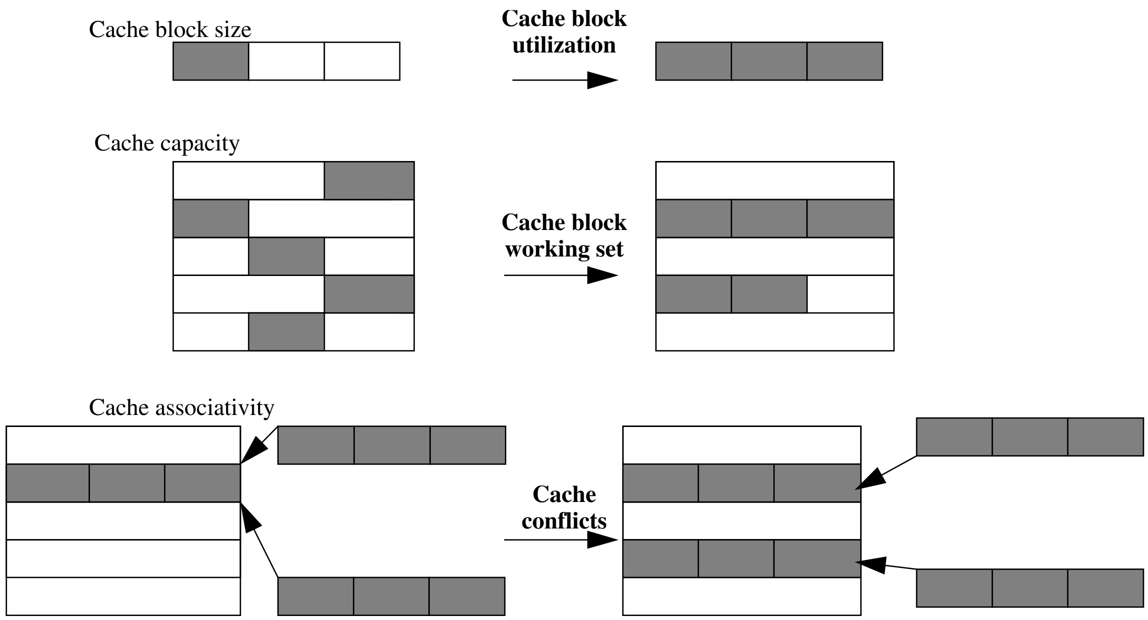

Figure 4. Improving cache performance (block size = 3 words, capacity = 5 blocks, associativity = 1). (a) Cache block utilization can be improved by packing cache blocks with contemporaneously accessed data. (b) Cache block working set can be reduced by a similar technique applied across multiple cache blocks. (c) Cache conflicts can be reduced by having contemporaneously accessed data map to different cache locations.

poraneously. Clustering improves spatial and temporal locality and provides implicit prefetching.

To illustrate the concept let us consider a tree structure. An effective way to cluster a tree is to pack subtrees[1] into a cache block. (Figure 8 shows a subtree-clustered binary tree.) An intuitive justification for binary subtree clustering is as follows. 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 greater than 2 for $k > 3$. Consider the alternative of a depth-first clustering scheme, in which the k nodes in a block form a single parent-child-grandchild-... chain. Here the expected number of accesses to the block per tree search is bounded by 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. In addition, tree modifications can destroy locality. However, our experiments indicate that for trees that change infrequently, subtree clustering is far more efficient than allocation-order clustering, which places contemporaneously allocated tree nodes in the same cache block.

2.2 Coloring

Caches have finite associativity, which means that only a limited number of concurrently accessed data elements can map to the same cache block without incurring conflict misses. Coloring maps contemporaneously accessed elements to non-conflicting regions of the cache. Figure 5 illustrates a 2-color scheme for a 2-way set-associative cache (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 (i.e., such that they do not conflict with each other), and the remaining elements are mapped to the other region. The mapping ensures that heavily accessed data structure elements do not conflict among themselves and

1. The term subtree is used to refer to subtree regions rather than complete subtrees.

are not replaced by infrequently accessed elements. In addition, if the gaps in the virtual address space that implement coloring correspond to multiples of the virtual memory page size, this scheme does not waste any physical memory.

2.3 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 compressed information. However, with high memory access costs, computation may be cheaper than additional memory references. Structure compression techniques include *data encoding techniques*, such as key compression [7], and *structure encoding techniques*, such as pointer elimination and hot/cold structure splitting.

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 eliminating the internal subtree pointers from the clusters in the tree shown in Figure 8.

Hot/cold structure splitting is based on the observation that most searches examine only a portion of individual elements until a match is found. Structure splitting separates heavily accessed (hot) portions of data structure elements from rarely accessed (cold) portions (see Figure 7b). This slightly increases the total size of the data structure, but can significantly reduce the size of the hot "working set."

3 Strategies for Cache-Conscious Data Placement

While cache-conscious pointer structure design offers significant performance benefits, there are several reasons why this approach is difficult for the average programmer to apply to real programs. First, application of these design principles is dependent on the data structure and its associated access pattern, and consequently requires complete understanding of an application's code and data structures. Moreover, they require knowledge of the underlying cache architecture—something many programmers are unfamiliar with. Finally, they require significant rewriting of an application's code.

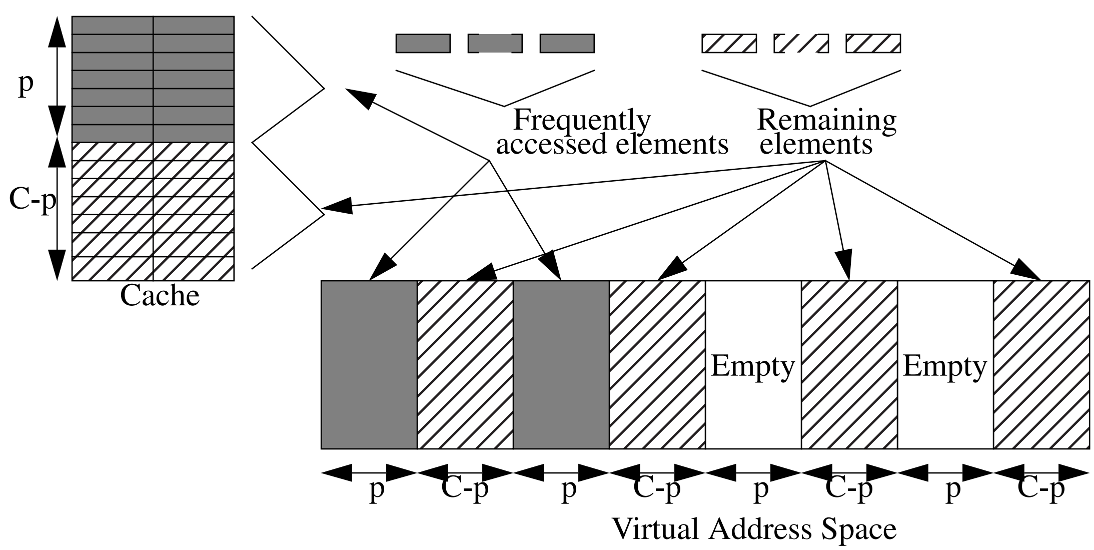

Figure 5. Coloring data structure elements to reduce cache conflicts. Frequently accessed elements are uniquely mapped to a portion of the cache so that they are not displaced by infrequently accessed elements. This is accomplished by inserting gaps in the virtual address space that are kept empty.

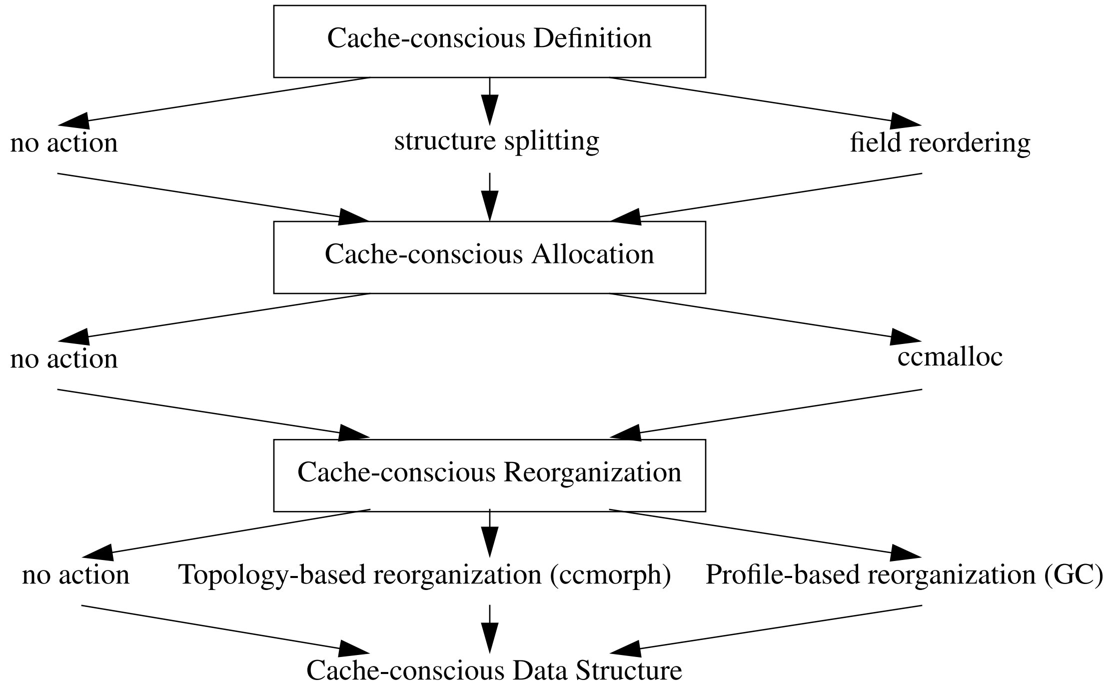

Figure 6. Using cache-conscious strategies to produce a cache-conscious data structure. The different cache-conscious strategies can be combined in a wide variety of ways to produce a cache-conscious data structure.

To address these problems, we have designed and evaluated several mostly-automatic and completely-automatic strategies for producing cache-conscious pointer structures. All these strategies have the common goal of making the benefits of cache-conscious data structures available to an average programmer, just as compilers provided programmers, unwilling or unable to code in assembly language, the ability to write high-performance programs. The different strategies apply the design principles described in the previous section to transform existing pointer structures into cache-conscious versions.

We choose to organize the various cache-conscious strategies according to the time—definition, allocation, or access time—when they are applied to data structures. Cache-conscious pointer structures can be constructed by changing the structure definition, by modifying the allocation policy for structure elements, or by reorganizing the structure layout. Changing a structure's definition by reordering fields permits clustering fields that are accessed contemporaneously in the same cache block. Splitting structures into a hot and cold portion based on program accesses permits packing more hot instances, that are accessed together, in the same cache block. Both of these techniques increase cache block utilization. Cache-conscious allocation attempts to co-locate contemporaneously accessed data elements in the same physical cache block at allocation time. This improves cache performance by increasing cache block utilization. Finally, cache-conscious reorganization attempts to transform the memory layout of pointer structures by linearizing them with respect to the expected data access pattern, and by mapping structure elements to reduce cache conflicts. The expected access pattern can be obtained from program profiles. For certain pointer structures such as trees, access information can be gleaned from data structure topology.

Figure 6 presents a flowgraph that illustrates how the various cache-conscious strategies described later in this section can be combined to produce a cache-conscious data structure.

3.1 Cache-Conscious Structure Definition

This section focuses on the internal organization of structure elements and explores two cache-conscious definition techniques—*structure splitting* and *field reordering*—that can improve the cache

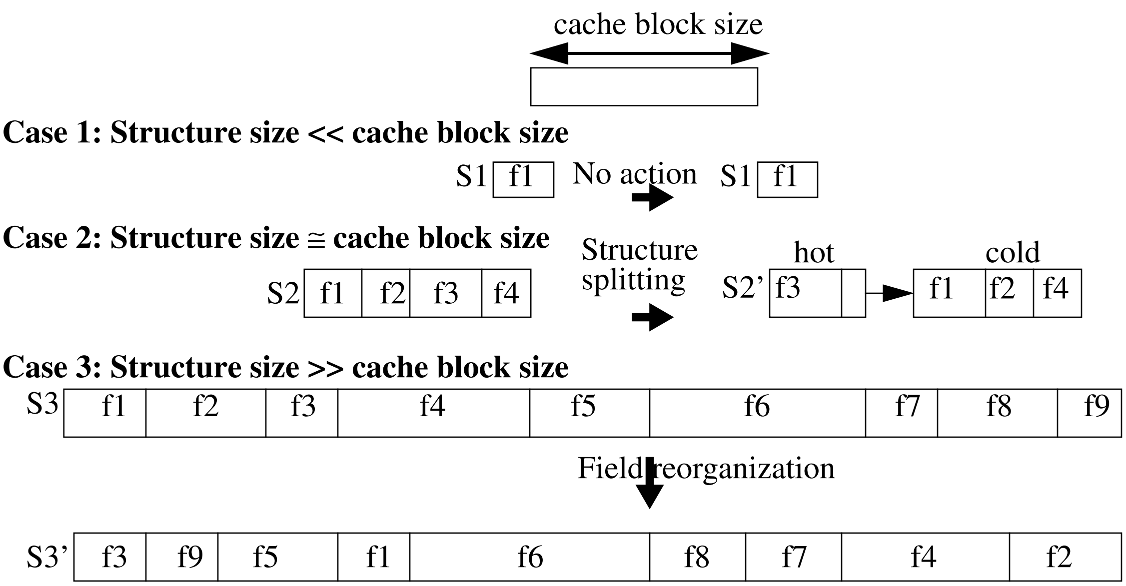

Figure 7. Cache-conscious structure definition. In case 1, the structure has a single field and is much smaller than the cache block size so no action is necessary at definition time. In case 2, where the structure size is comparable to the cache block size, splitting may permit multiple hot structure instances to be packed 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.

behavior of programs. Figure 7 illustrates the relationship of cache-conscious definition technique to the size of structure instances. As the figure indicates, there are three possibilities, depending on the size of structure instances relative to the cache block size. The remainder of this section describes these techniques in more detail.

3.1.1 Structure Splitting

Many Java objects are comparable to the size of a cache block (case 2) [5]. In addition, since Java is a type-safe language, class (structure) splitting can be automated. The first step in this process is to identify class member fields as hot (frequently accessed) or cold (rarely accessed). While it may be possible to classify some member fields via static analysis, we profile a program to determine member access frequency since this appears to be a simpler and more general approach. A compiler extracts cold fields from the class and places them in a new object, which is referenced indirectly from the original object. Accesses to cold fields require an extra indirection to the new class, while accesses to hot fields remain unchanged. The overhead of splitting 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. The splitting algorithm takes these factors into account and is carefully designed to reduce these costs (see [5] for details). In addition, we use our garbage collection scheme for cache-conscious object co-location (see Section 3.3.2) to aggressively exploit the advantage offered by smaller (hot) class instances by packing more hot instances in the same cache block.

For five medium-sized Java benchmarks (these include javac, a bytecode compiler, and javadoc, a document generator), class splitting combined with our garbage collection scheme for cache-conscious object co-location reduced L2 cache miss rates by 29–43%, with class splitting accounting for 26–62% of this reduction, and improved performance by 18–28%, with class splitting contributing 22–66% of this improvement [5]. These experiments were run on a single processor of a 167MHz 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 be used for programmer feedback.

3.1.2 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 is likely to produce hot instances that are larger than a cache block, making it ineffective. Reordering structure fields to place those with high temporal affinity in the same cache block can improve cache block utilization. Typically, fields in large structures are grouped conceptually, which may not correspond to their temporal access pattern. Unfortunately, the logical order for a programmer may cause structure references to interact poorly with a program's data access-pattern and result in unnecessary cache misses. Compilers for many languages are constrained to follow the programmer-supplied field order and so cannot correct this problem.

To investigate the benefits of field reordering, we implemented an algorithm for recommending reordering of structure fields in C programs. This field reordering 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. This data is used to construct a field affinity graph for each structure. These graphs are then processed to produce field order recommendations. Measurements on a 4 processor 400MHz Pentium II Xeon system with 1MB L2 cache, 4GB memory, and 200 7200rpm Clariion fibre channel drive disks, indicate that reordering fields in 5 active structures improves the performance of Microsoft SQL Server 7.0, a large, highly tuned commercial application, by 2–3% on the TPC C benchmark [5].

3.2 Cache-Conscious Structure Allocation

The elements of a data structure are typically allocated with little concern for a memory hierarchy. The resulting layout may interact poorly with the program's data-access patterns, thereby causing unnecessary cache misses and reducing performance. Cache-conscious allocation addresses this problem by attempting to co-locate contemporaneously accessed data elements in the same cache block. Since a heap allocator is invoked many times, it 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 also safe, in that incorrect usage only affects program performance, but not correctness.

`ccmalloc` is a memory allocator similar to `malloc`, but `ccmalloc` takes an additional parameter that points to an existing data structure element likely to be accessed contemporaneously with the element to be allocated (e.g., the parent of a tree node). `ccmalloc` attempts to locate the new data item in the same cache block as the existing item. The following code from the Olden benchmark *health* illustrates the approach:

```
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;
}
```

Our experience with `ccmalloc` indicates that even a programmer unfamiliar with an application can often select a suitable parameter by local examination of code surrounding the allocation statement and obtain good results.

In a memory hierarchy, different cache block sizes means that data can be co-located in different ways. `ccmalloc` focuses only on L2 cache blocks. In the Sun UltraSPARC 1 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 both more likely to be successful and to incur less overhead. On a system with a larger L1 cache block it would probably be advantageous to adopt a hierarchical approach with co-location first attempted in the same L1 cache block. If that fails the subsequent co-location attempt could be in the same L2 cache block.

Cache-conscious heap allocation (`ccmalloc`) resulted in a speedup of 27% for VIS, a 160,000 line system that formally verifies finite state systems using Binary Decision Diagrams (BDDs) [4]. These results were obtained on a 167MHz Sun Ultraserver E5000. Significantly, very few changes to the program (less than 300 lines of code) produced these large performance improvements. This indicates that cache-conscious data placement can even improve the performance of graph-like data structures (BDDs are directed acyclic graphs), in which data elements have multiple parents.

3.3 Cache-Conscious Structure Reorganization

A complementary approach to cache-conscious allocation is to reorganize a structure's memory layout to correspond to its access pattern. Unlike cache-conscious allocation, which can be done just once when a data element is created, cache-conscious reorganization can be done as often as required. General graph-like structures require a detailed profile of a program's data access patterns for successful memory layout reorganization [1, 6]. However, a very important class of structures (trees) possess topological properties that permit cache-conscious data reorganization without profiling. Section 3.3.1 presents a transparent (semantics-preserving) cache-conscious tree reorganizer (`ccmorph`) that applies the clustering and coloring techniques described in Section 2. `ccmorph` is appropriate for "read-mostly" data structures—one that are built early in a computation and subsequently heavily referenced. With this approach, neither the construction nor the consumption code need change, as the structure can be reorganized between the two phases. Moreover, if the structure changes slowly, `ccmorph` can be periodically invoked.

Languages that support garbage collection offer a more attractive alternative. Copying garbage collectors, which support automatic memory management, determine when dynamically allocated storage has become unreachable and automatically recycle that memory by traversing the heap and copying live data to a separate region of memory. All memory in the traversed space is then freed up for reuse. Thus, 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 that pointers be distinguished from non-pointer data. Hence, it cannot be implemented as described for low-level languages, such as C or C++, that support arbitrary pointer-manipulation operations and preclude transparent data movement, though new hardware mechanisms proposed by Luk and Mowry may remove this obstacle [9]. On the other hand, 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 be used to reorganize data and produce a cache-conscious structure layout.

3.3.1 Topology-based Structure Reorganization

In a language such as C, which supports unrestricted pointers, analytical techniques cannot precisely identify 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` transparently reorganizes a tree data structure to improve its locality by applying the clustering and coloring techniques from

Section 2.

ccmorph operates on tree-like structures that have homogeneous elements and do not have external pointers into the middle of the structure (or on any data structure that can be decomposed into components satisfying this property). However, it has a liberal definition of a tree in which elements may contain a parent or predecessor pointer. A programmer supplies ccmorph with a pointer to the root of a data structure, a function to traverse the structure (*next_node*), and cache parameters. For example, the following code is used to reorganize the quadtree data structure in the Olden benchmark *perimeter.* The programmer supplies 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);
   }
}
```

ccmorph copies a structure into a contiguous block of memory (or a number of contiguous blocks for large structures). In the process, it partitions a tree-like structure into subtrees that are laid out linearly (see Figure 8).The structure is also colored 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. ccmorph determines the values of *p* and the size of subtrees from the cache parameters and the structure element size. In addition, it takes care to ensure that the gaps in the virtual address space that implement coloring correspond to multiples of the virtual-memory page size.

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 to be modeled. Cache-conscious clustering and coloring of the octree produced a speedup of 42% (this includes the overhead of restructuring the octree) on a 167MHz Sun Ultraserver E5000 system [4].

3.3.2 Profile-based Structure Reorganization Using Garbage Collection

A cache-conscious data layout places objects with high temporal affinity near each other, so that they are likely to reside in the same cache block. In this approach, a program is instrumented to profile its

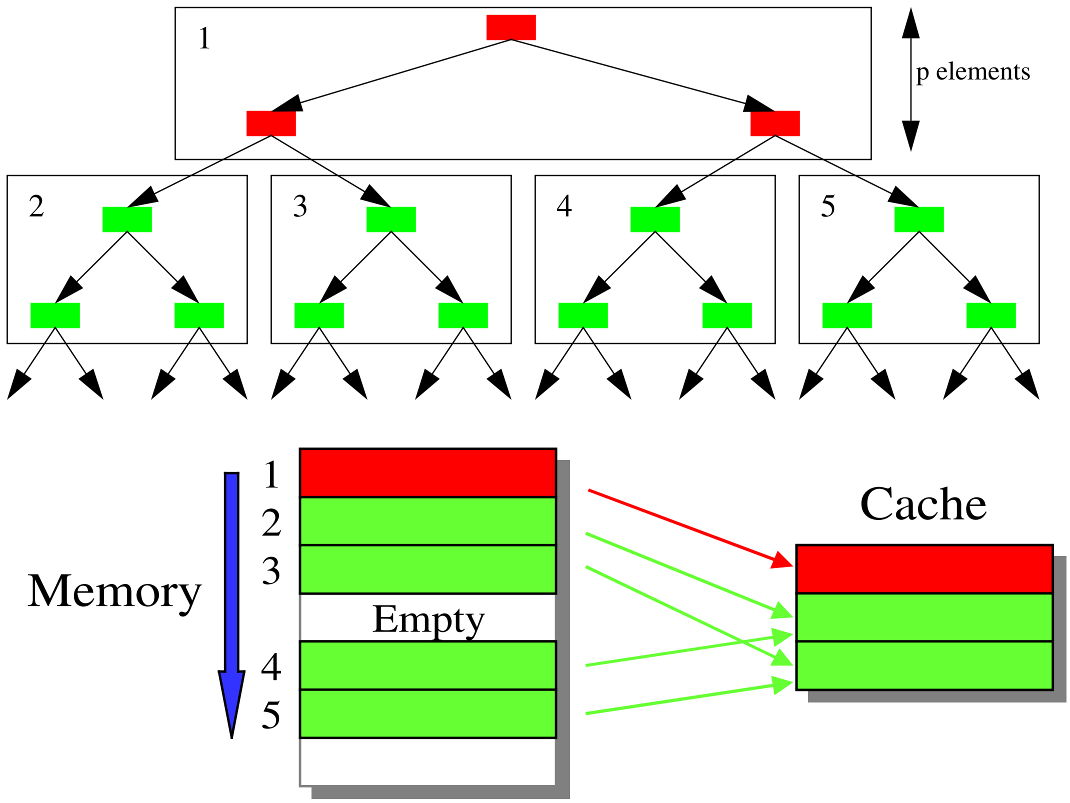

Figure 8. Cache-conscious tree reorganization. Subtree-clustering is applied to the entire tree. The top-levels (frequently accessed elements) of the tree are allocated such that they map to a portion of the cache where they cannot be displaced by lower-level tree nodes (infrequently accessed elements).

data access patterns. The profiling data gathered during an execution is used 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 [5, 6]. The garbage collector uses this profile to construct an object affinity graph, in which weighted edges encode the temporal affinity between objects (nodes). A new copying garbage collection algorithm makes a depth-first traversal of the affinity graph to produce cache-conscious data layouts while copying objects. The technique is completely 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 16–42% and improves program performance by 10–37% (including the real-time data profiling overhead) [5, 6]. Further, we compared our cache-conscious copying scheme against an earlier algorithm that attempted to improve a program's virtual memory (page) locality by changing the traversal algorithm used by a copying garbage collector (the Wilson-Lam-Moher algorithm [11]). The results show that our cache-conscious object layout technique reduces cache miss rates by 14-41%, and improves program performance by 8–31% over their technique, which indicates that page-level improvements are not necessarily effective at the cache level [6]. These experiments were run on a single processor of a 167MHz Sun Ultraserver E5000 system.

4 Conclusions

Traditionally, pointer-based data structures were designed and programmed as if memory access costs were uniform. Increasingly expensive memory hierarchies have falsified this simplifying assumption and opened an opportunity for significant performance improvements with redesigned data structures that use caches more effectively. This paper discusses three data placement design principles—*clustering*, *compression*, and *coloring*—that programmers can use to improve the spatial and temporal locality of pointer-based data structures.

However, the design of cache-conscious data structures requires a deep understanding of a program's structures and operation, and familiarity with a machine's memory architecture. These prerequisites may limit the use of cache-conscious data structures to performance critical portions of code written by expert programmers, much as assembly programming is used today. To make the performance benefits of cache-conscious structures available to the average programmer, this paper presents several effective strategies that reorder the internal layout of a structure's fields and arrange structure instances in memory. While some of these techniques require modest programmer assistance for low-level languages such as C and C++, others are completely automatic. Measurements show that these techniques reduce cache miss rates and achieve significant performance improvements (up to 40%) on real programs.

Based on past trends and future technology, the processor-memory performance gap will continue to increase and software will continue to grow larger and more complex. Although the algorithmic and data structure design phase of software development is the first, and perhaps best, place to address this growing gap, the complexity of software design, and an increasing tendency to build large software systems by gluing together smaller components, does not favor a focused, integrated approach. These realities make techniques for producing cache-conscious data layouts, such as those presented in this paper, an essential aspect of the process of achieving the highest performance on current and future machines. In addition, given the restrictions low-level languages place on generating cache-conscious data layouts, these techniques may help narrow, or even reverse, the performance gap between high-level programming languages, such as Lisp, ML, or Java, and low-level languages, such as C and C++.

Acknowledgments

Most of this research was conducted as part of Chilimbi's Ph.D. thesis at the University of Wisconsin-Madison [3]. This work is supported in part by the National Science Foundation (MIPS-9625558, CCR-9357779, EIA-9971256, and CDA-9623632), Microsoft Corporation, and Sun Microsystems. We especially thank members of the Wisconsin Wind Tunnel project, Craig Chambers and Dave Grove for the Vortex compiler infrastructure, 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 many useful comments on an earlier draft of this paper.

References

[1] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. "Cache-Conscious Data Placement." In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 139–149, Oct. 1998.

[2] 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.

[3] Trishul M. Chilimbi. "Cache-Conscious Data Structures—Design and Implementation." *Ph.D. thesis*, University of Wisconsin-Madison, 1999.

[4] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. "Cache-Conscious Structure Layout." In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI)*, May 1999.

[5] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. "Cache-Conscious Structure Definition." In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI)*, May 1999.

[6] Trishul M. Chilimbi and James R. Larus. "Using Generational Garbage Collection to Implement Cache-Conscious Data Placement." In *Proceedings of the International Symposium on Memory Management*, pages 37–48, October 1998.

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

[8] 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.

[9] Chi-Keung Luk and Todd C. Mowry. “Memory forwarding: Enabling aggressive layout optimizations by guaranteeing the safety of data relocation.” *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA ‘99)*, June 1999.

[10] 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.

[11] 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*.

[12] 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*.