Cache-Conscious Data Structures — Design and Implementation

by

Trishul Madhukar Chilimbi

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN—MADISON

1999

Abstract

The increasingly expensive cost of accessing memory provides an opportunity to significantly improve the performance of computer programs by redesigning their data structures to use processor memory caches more effectively. This thesis explores principles for designing cache-conscious data structures, such as clustering, coloring, and compression. These techniques improve the spatial and temporal locality of pointer-based data structures. To formalize the design process, this thesis provides an analytic framework that quantifies the performance of cache-conscious pointer structures.

Unfortunately, cache-conscious structure design requires a deep understanding of a program's structures and operation, and familiarity with a machine's memory architecture. These prerequisites can limit cache-conscious structures to performance-critical portions of code written by expert programmers, much as assembly code is used today. To make the performance benefits of cache-conscious structures available to the average programmer, this thesis investigates several techniques that facilitate the creation of cache-conscious structure layouts.

Ideally, cache-conscious structure transformations should require no programmer intervention. However, low-level languages, such as C and C++, make this goal impossible to attain without hardware support. While offering no silver bullet for these low-level languages, this thesis explores several mechanical techniques that greatly reduce the programming effort and application knowledge required to improve cache performance. These techniques produce a cache-conscious arrangement of structure instances in memory. Additional techniques manipulate the internal organization of fields in a structure instance to make the layout cache-conscious. Finally, the thesis shows that these cache-conscious techniques can be packaged into easy-to-use tools.

This thesis also describes a more attractive alternative for languages that support garbage collection. In this situation, a generational garbage collector can easily be modified to produce cache-conscious layouts of small objects. The thesis demonstrates the feasibility of low-overhead, real-time profiling of data access patterns for object-oriented languages and describes a new copying algorithm that uses this information to produce cache-conscious object layouts. Measurements show that this technique reduces cache miss rates and improves program performance significantly. Techniques such as these 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 or C++.

Acknowledgements

Firstly, I would like to thank my advisor Jim Larus. Jim always let me pursue my ideas, even if he thought some of them misguided. His healthy skepticism often spurred me on. In addition, his insight and experience saved me much effort. Jim always made the time to provide me with extensive feedback. While I failed to emulate his time-management skills, they served as an inspiration.

Next, I would like to thank Mark Hill. Mark set aside large amounts of time for stimulating discussions. Both Mark and Jim taught me the importance of not only doing good research but also taking the time to present it clearly.

I owe much to the Wisconsin Wind Tunnel project, co-supervised by Mark Hill, Jim Larus, and David Wood. Being part of a large, top-notch research project taught me the value of collaboration as well as the effort needed to achieve it.

I would also like to thank Tom Reps. Tom was very encouraging and spent large amounts of time attending my talks and reading my research. He provided me with much useful feedback. I also learned a lot about sailing from him.

I would also like to thank my many friends, especially those at Hoofers Sailing Club, who made my stay in Madison a most enjoyable one. Though you probably delayed my graduation, it was time well spent.

I owe much to my parents, Madhukar and Romila Chilimbi. Finally, I would like to thank Anne Doremus for her love and support. In addition, she spent copious amounts of time proof-reading my dissertation despite all the jargon.

Table of Contents

List of Figures

To think is to confine yourself to a
single thought that one day stands
still like a star in the world's sky.
—Martin Heidegger, Poetry, Language, Thought

Chapter 1

Introduction

A computer system consists of processors for performing computation on data, a storage system for data, and a variety of peripheral devices for interacting with the external environment. To address the widely varying data requirements of different applications, storage systems have evolved into primary or main memory, secondary storage, and tertiary storage. These systems possess decreasing costs and data access speeds, and increasing storage capacities. Main memory, which like microprocessors is implemented using semiconductor technology, has the fastest data access speed, but it is the most expensive and has the smallest data capacity.

Speed of a computer system, which is calibrated in terms of time taken to execute stored programs, is an important consideration. As with any complex system, it is limited by the speed of its slowest component along the critical execution path. For many programs, this is the speed of the secondary storage system, as tertiary storage is almost exclusively used for archiving data and very rarely accessed during program execution. However, advances in main memory technology have increased data capacity to the point where many applications seldom access secondary storage, and the limiting factor is the speed of main memory. For applications that frequently access secondary storage, such as databases, much research has focused on optimizing the references to secondary storage. Consequently, even these applications have significant portions of computation that are memory limited.

Since both microprocessors and memory are implemented with semiconductor technology, one would expect their speeds to be comparable. However, the speed of microprocessors has increased 60% per year for almost two decades. Yet, over the same period, memory access time decreased by less than 10% per year [56]. These trends appear likely to persist barring an unforeseen technological breakthrough. This is because the primary driving force behind memory technology is storage capacity, and current technology precludes the manufacture of high-capacity memories with fast access times.

The unfortunate, but inevitable, consequence is an ever-increasing processor–memory performance gap. Memory caches are the ubiquitous hardware solution to this problem. These are small, fast memories that store recently accessed data items and attempt to intercept and satisfy data requests without accessing main memory. In the beginning, a single level of cache sufficed, but the increasing performance gap (now almost two orders of magnitude) requires a hierarchy of caches, which results in a wide range of memory access costs.

Caches can improve performance by exploiting data reference locality (see Figure 1-1). 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.

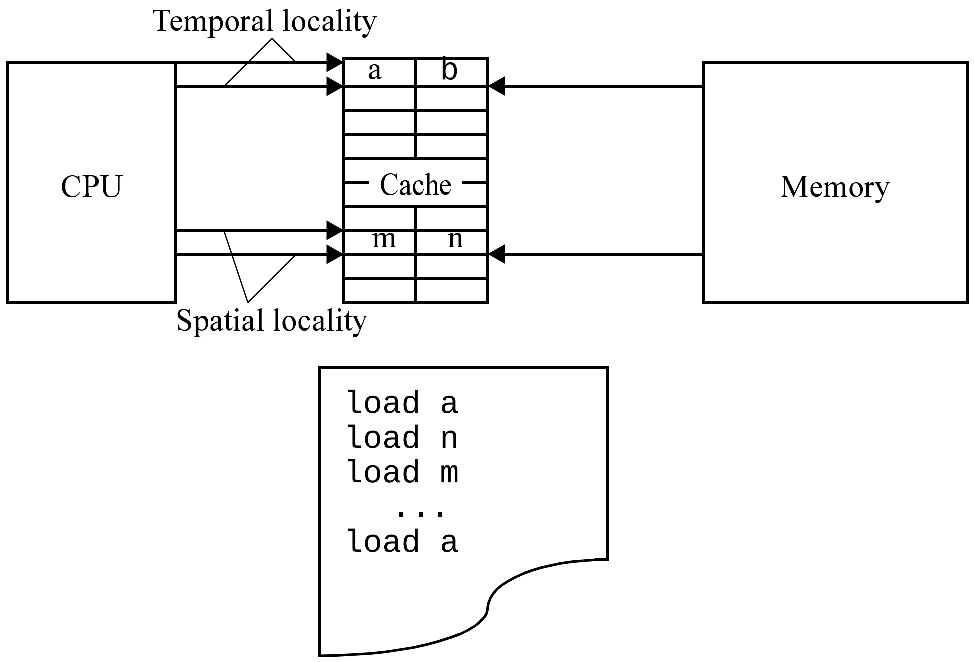


Figure 1-1. Exploiting locality with a cache.

In addition to caches, a variety of hardware and software techniques—such as prefetching [53, 13], multithreading [46, 68], non-blocking caches [41], dynamic instruction scheduling [75], and speculative execution [69]—have been developed and implemented to reduce or tolerate memory latency. Prefetching attempts to reduce latency by anticipating data that will be required, and fetching it into the cache before it is requested. Multithreading tolerates latency by context switching from a stalled thread to another runable thread. A non-blocking cache can tolerate multiple outstanding misses and return data in a different order from the requests. In dynamic instruction scheduling, the hardware rearranges instructions and executes them out-of-order to reduce stalls, but guarantees instructions are committed in-order. Speculative execution predicts a program's control flow and executes instructions eagerly. If a prediction is incorrect, the misspeculated instructions are squashed.

Despite these techniques, which require complex hardware or software, many programs' performance is dominated by memory references. Moreover, high and variable memory

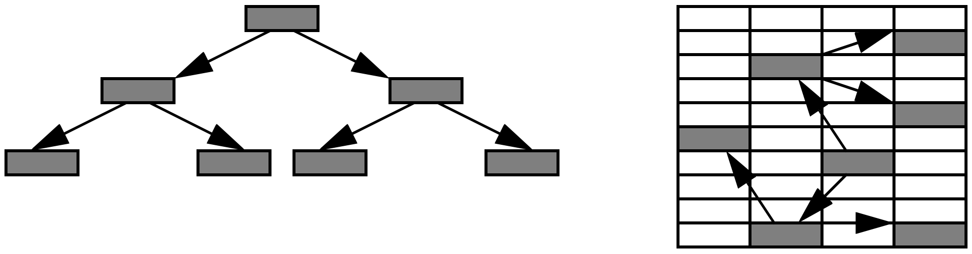


Figure 1-2. Pointer structures and their memory layout.

access costs undercut the fundamental random-access memory (RAM) model 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 databases and operating systems, make extensive use of pointer structures to store data.

A pointer structure is a collection of heap-allocated data elements connected by pointers. Figure 1-2 illustrates a tree pointer structure. Since pointer structures are collections of heap allocated elements, they can be dynamically sized while a program is executing. A pointer structure can be grown by allocating additional elements on the heap, and attaching them to the existing structure with pointers. Shrinking a structure is the reverse process in which elements are deleted and the freed heap space made available for reuse. Pointer structures are frequently used in applications in which the data requirements are not known until the program executes or varies widely during execution. Due to their dynamic nature and reliance on heap-allocated storage, pointer structures tend to have less regular access patterns than array structures.

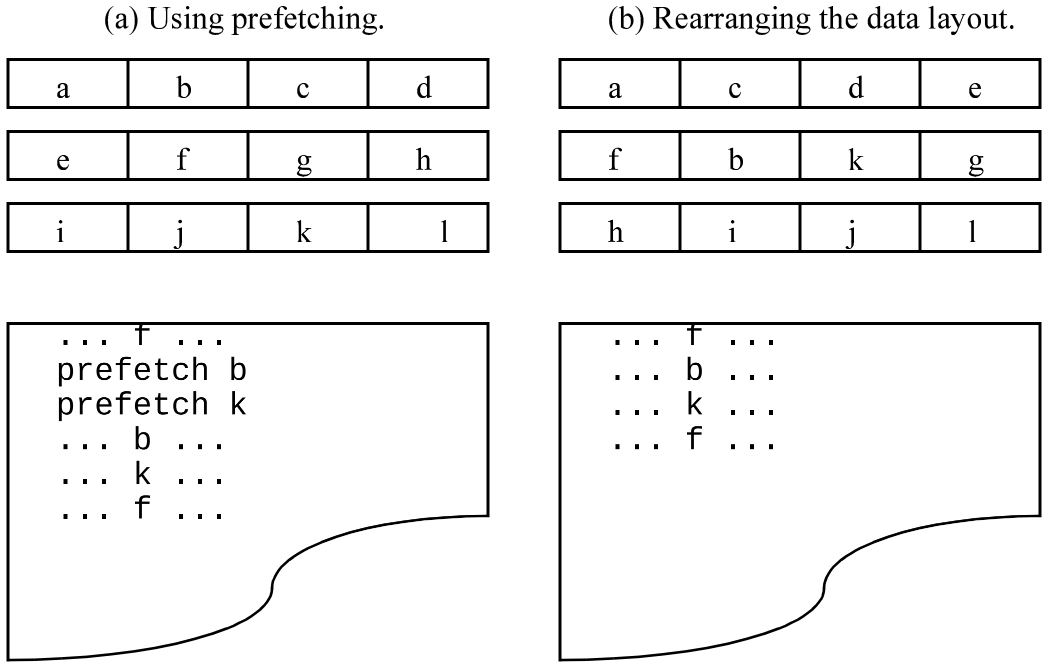
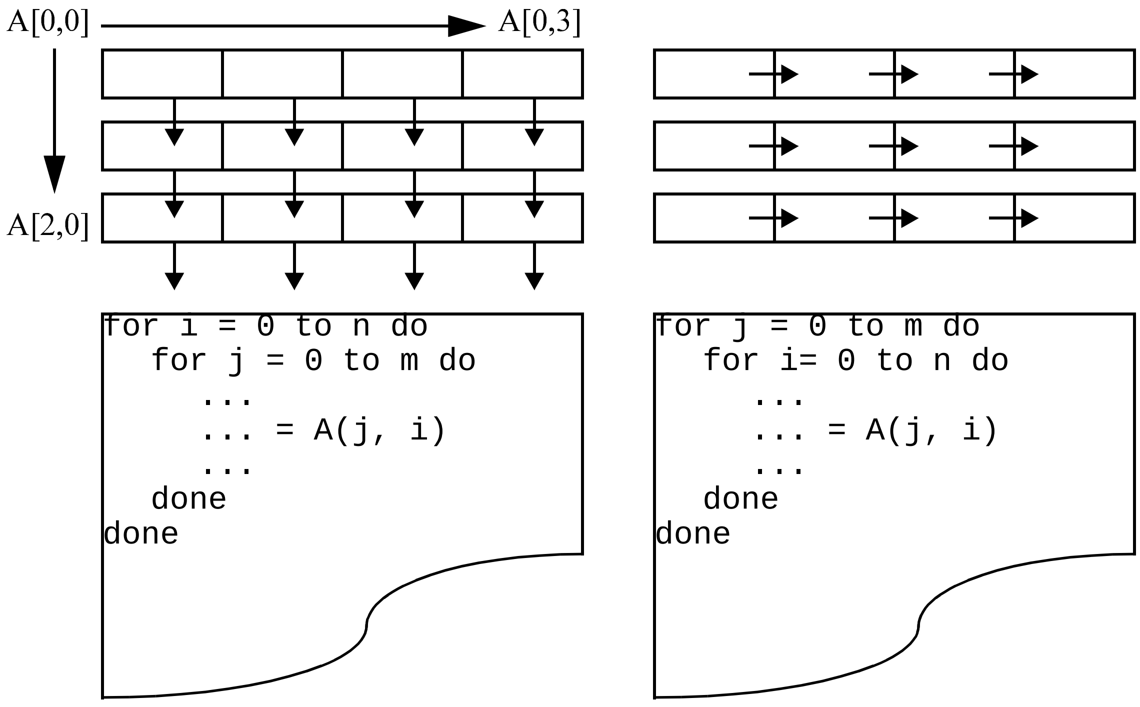


Figure 1-3. Different approaches to reducing memory latency.

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 [11, 57]. 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). Figure 1-3 illustrates the differences between these two approaches. The prefetching approach, which focuses on alleviating memory latency, can increase memory bandwidth requirements if the prefetched data is replaced from the cache before it is referenced. This can occur if the cache blocks that contain data items *b* and *k* map to the same location in the cache. In addition, the prefetch instructions, which incur an address translation overhead, must be repeated at all program points where that access sequence occurs. By contrast, an approach that rearranges the data layout to improve reference locality, suffers from none of these problems.



```
for i = 0 to n do
  for j = 0 to m do
    ...
    ... = A(j, i)
    ...
  done
done
```

```
for j = 0 to m do
  for i= 0 to n do
    ...
    ... = A(j, i)
    ...
  done
done
```

Figure 1-4. Improving software locality through program transformation.

In general, software 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 [84, 14, 28]. Figure 1-4 illustrates this with an example. The code snippet on the left references array elements in a manner that cycles through different cache blocks, accessing a single element in a cache block at a time. 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. 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.

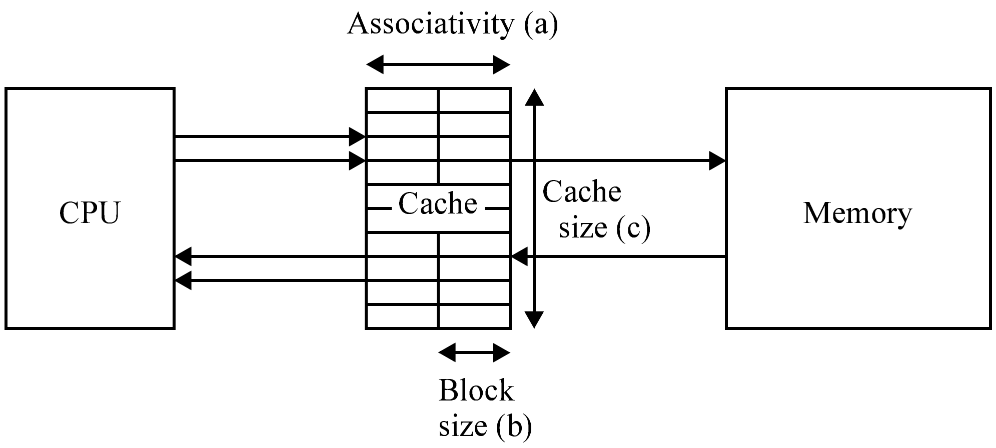


Figure 1-5. Memory cache.

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.

1.1 Cache Performance

As described earlier, caches are small, fast memories that store recently referenced data (see Figure 1-5). 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 that encompass multiple words (typically 16-128 bytes). For several reasons, caches have finite associativity (typically 1, 2, or 4) and this restricts where a block can be placed in the cache.

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 poor cache performance. The average memory-access time for a machine architecture with a cache is given by

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

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.

Hill characterized cache misses as compulsory misses, capacity misses, and conflict misses [29]. 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 blocks mapping to the same position in the cache.

Figure 1-6 illustrates different approaches to improving cache performance. The shaded units indicate contemporaneously accessed data items. Since data is transferred in cache block sized units that can contain multiple data items, increasing cache block utilization by placing contemporaneously accessed data in the same cache block increases the efficiency of data transfer from memory and provides an implicit prefetching mechanism. This reduces the number of compulsory and capacity misses. Moreover, making more efficient use of cache space by reducing a structure's cache block footprint will also reduce the number of capacity misses (see Figure 1-6). Finally, mapping concurrently accessed structure elements (which do not fit in a single cache block) to non-conflicting cache blocks reduces conflict misses.

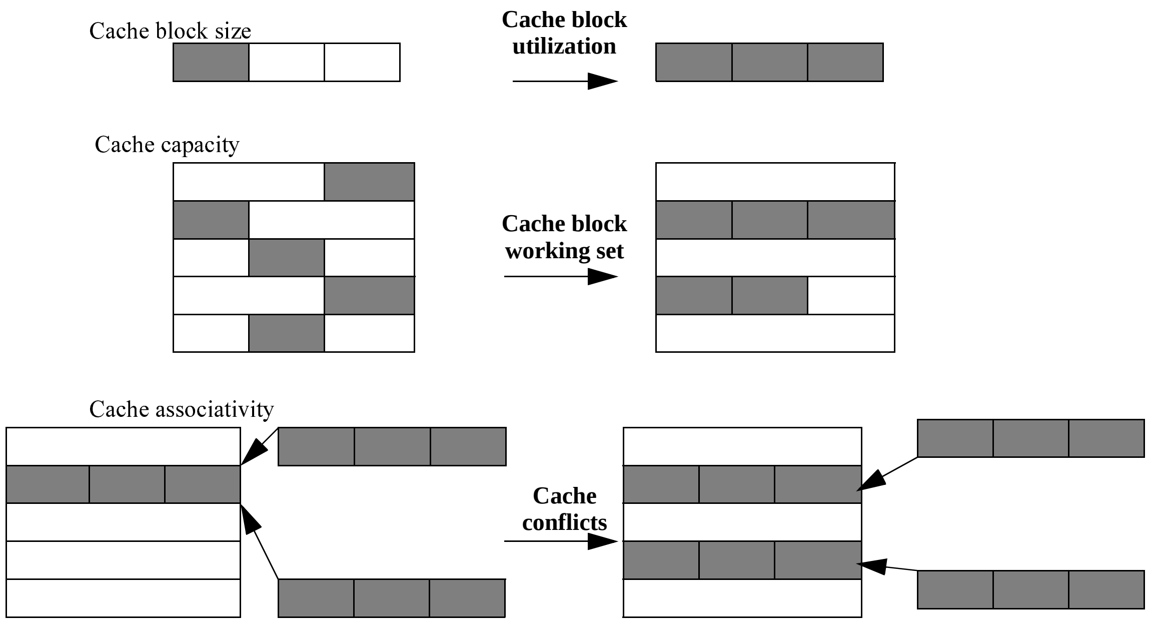
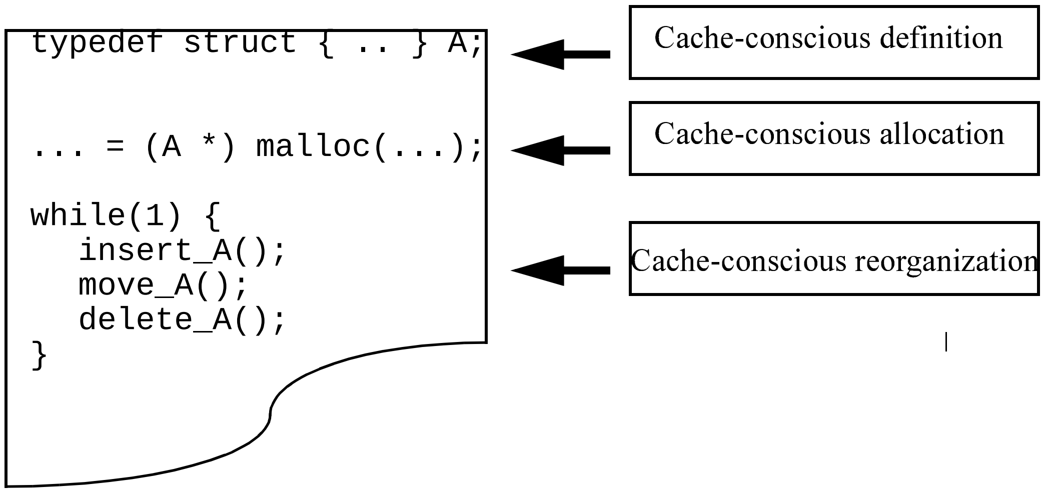


Figure 1-6. Improving cache performance.

1.2 Improving the Cache Locality of Pointer Manipulating Programs

This thesis explores several design principles that improve cache performance by increasing a pointer 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. This increases cache block utilization and reduces the cache block working set (see Figure 1-6). *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. This thesis explores applying these placement design principles to construct cache-conscious data structures that demonstrate significant performance gains.

To formalize the cache-conscious design process, this thesis presents an analytic framework that quantifies the performance benefits of cache-conscious pointer structures. A key part of this framework is a data-structure-centric cache model of a series of accesses that


Cache-conscious definition
Cache-conscious allocation
Cache-conscious reorganization

```
typedef struct { .. } A;

... = (A *) malloc(...);

while(1) {
   insert_A();
   move_A();
   delete_A();
}
```

Figure 1-7. Strategies for cache-conscious data placement.

traverse a pointer structure. The performance of a pointer structure is characterized by its amortized miss rate over a sequence of such accesses. The model is applied to cache-conscious pointer structures and its predictions validated.

While cache-conscious pointer structure design offers significant performance benefits, there are several reasons why this approach is difficult to apply to real programs. First, applying 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. Next, they require knowledge of the underlying cache architecture—something many programmers are unfamiliar with. Finally, they require significant rewriting of an application's code. The rest of the thesis addresses these issues.

To address the problems of architectural familiarity and extensive application rewriting that make it impractical to apply cache-conscious design principles to large legacy applications, we have designed and evaluated several mostly-automatic and completely-automatic strategies for implementing cache-conscious pointer structures. As Figure 1-7 illustrates, cache-conscious pointer structures can be implemented 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 pointer structure layouts by linearizing them with respect to the expected data access pattern, and 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. For a tree structure, a node access is likely to be followed by an access to a child of that node. Hence clustering tree nodes into subtrees that fit in a cache block increases cache block utilization. In addition, the top levels of a tree are accessed much more frequently than the bottom levels. Mapping a tree such that the bottom levels of the tree do not conflict with the top levels in the cache will reduce cache conflicts.

The programming language employed can dramatically affect the feasibility and ease of applying these cache-conscious transformation strategies. For example, languages such as C and C++ support type-casting and pointer arithmetic operations that make it extremely difficult to transparently move structures (i.e., without programmer intervention), and hamper any structure reorganization strategy. In addition, these languages do not hide the internal representation of structure instances and this hinders transparently modifying the structure definition. Despite these problems, this thesis explores the application of each of the cache-conscious transformation strategies—*cache-conscious definition*, *cache-conscious allocation*, and *cache-conscious reorganization*—that require minimal programmer assistance in such unfriendly environments, and yet produce large performance improvements.

On the other hand, modern languages such as Java provide a much more conducive environment for such data layout optimizations. The unrestricted pointers of C and C++

are replaced by references that facilitate transparent movement of structures. In fact, automatic memory-management schemes such as copying garbage collection, commonly employed by these languages, routinely move data. This thesis shows that copying garbage collection can be used for transparent and automatic implementation of cache-conscious data layouts. In addition, modern languages such as Java are type-safe, and this permits transparently changing the internal structure representation. We exploit this to automatically split structures into a hot and cold portion. This splitting permits more hot structure instances, which are referenced together, to be packed into a cache block. When combined with cache-conscious garbage collection, this scheme produces significant speedups.

In summary, this thesis explores a wide variety of highly effective approaches for improving the cache performance of pointer-manipulating programs both in the context of low-level languages such as C and C++, as well as for modern languages such as Java. However, given the relative ease of applying data layout optimizations to Java as compared to C/C++, and the growing processor-memory performance gap, we believe that such data layout optimizations may enable large Java programs to approach and perhaps eventually surpass the performance of equivalent applications written in C/C++.

1.3 Overview of the Dissertation

Chapter 2 explores the role of clustering, coloring, and compression in cache-conscious pointer structure design. It presents a brief overview of cache architectures and discusses the impact of cache parameters on structure design. Tree pointer structures are used to illustrate the cache-conscious design process. The large performance benefits of these cache-conscious pointer structures are established experimentally.

Chapter 3 presents an analytic framework for quantifying the performance improvements that result from cache-conscious data layouts. The framework is applied to cache-conscious pointer structures, and its predictions are validated against experimental results. The framework is shown to have good predictive power, underestimating the actual performance improvements by not more than 15%, and accurately predicting the shape of speedup curves.

Chapter 4 explores two strategies—*cache-conscious allocation* and *cache-conscious reorganization*—for implementing cache-conscious pointer structure layouts in unfriendly environments such as C and C++, where moving structures is in general unsafe. These strategies produce a cache-conscious arrangement of structure instances in memory. The chapter describes tools—`ccmalloc` and `ccmorph`—that embody each of these strategies. It illustrates the possibility of cache-conscious allocation by describing a solution wherein the heap allocator—`ccmalloc`—takes an additional, programmer-supplied parameter, which is a pointer to a data structure element that is likely to be accessed contemporaneously, and attempts to co-locate the new data item in the same physical cache block as the existing data. The chapter investigates alternative strategies to handle the case where the selected cache block is full. This technique has the advantage that an incorrect choice for the programmer-supplied parameter does not affect a program's correctness. For small pointer benchmarks, `ccmalloc` produced performance improvements of 12%—194%, and a large, real-world application improved by 27%.

In addition, this chapter investigates the task of reorganizing pointer structure layout during program execution. Two possibilities exist for such cache-conscious data reorganization. While general, graph-like structures require a detailed profile of a program's data-access patterns for successful reorganization, a very important class of structures (trees) possess topological properties that permit cache-conscious data reorganization without profiling. The chapter describes a topology-based reorganizer—`ccmorph`—that transparently transforms tree-like structures with homogeneous elements, and without external pointers into the middle of the structure. `ccmorph` exploits two properties of tree topology for this reorganization. First, for random key searches, packing subtrees into cache blocks is optimal. Second, a tree can be placed in memory such that frequently accessed structure elements (top levels of a tree), and infrequently accessed elements (all other levels of a tree) are mapped to non-conflicting cache regions. `ccmorph` improved the performance of several small pointer benchmarks by 28%–138%, and produced a 42% speedup for a realistic application.

Chapter 5 explores cache-conscious layout transformations in the context of modern languages such as Java that provide a much more conducive environment for such data layout optimizations. These languages support automatic memory management using garbage collection. This chapter shows that copying garbage collection can be used for transparent and automatic implementation of cache-conscious data layouts. The chapter discusses a profile-based reorganization technique that uses a generational copying garbage collector to produce a cache-conscious data layout, in which objects with high temporal affinity are placed next to each other, so that they are likely to reside in the same cache block. First, we discuss a technique for collecting profiling information about data-access patterns in object-oriented languages—in real-time and with low overhead (< 6%)—that exploits the observation that most objects are small (less than 32 bytes). Next, a new copying algorithm utilizes this profile information to produce a cache-conscious object layout. Experimental results for several object-oriented programs show that our cache-conscious data placement technique reduces cache miss rates by 16–42% and improves program performance by 10–37% over the traditional (Cheney's) copying algorithm. In addition, it outperformed a page-conscious copying algorithm (Wilson-Lam-Moher) by 8%–31%, indicating that improving locality at the page level is not necessarily beneficial at the cache level.

The effectiveness of such cache-conscious object co-location depends on the average program object size, and it does not perform as well for programs that manipulate larger objects (for e.g., many Java programs). To address this difficulty, Chapter 6 investigates structure definition strategies for manipulating the internal organization of fields in a structure instance to make the layout cache conscious. It describes a technique for partitioning structures into a hot and cold portion using field-access statistics from program profiles. This partitioning technique exploits the observation that for many programs, most of their structure references are often to the same, small set of fields, and permits more hot structure instances, which are referenced together, to be packed into a cache block. For five medium-size Java benchmarks (3000–28,000 lines), hot/cold object partitioning combined with cache-conscious object co-location reduced cache miss rates by 29–38%, with field partitioning accounting for 7–21% of the reduction, and improved performance by 18–31%, with field partitioning contributing 5–22%.

In addition, Chapter 6 explores the possibility of defining pointer structures in a cache-conscious manner by reordering structure fields. A suitable field order can improve a structure's cache block utilization and reduce its cache pressure. The chapter discusses an algorithm for structure field reordering and describes a tool—bbcache—that implements this algorithm. bbcache correlates static information about the source location of structure field accesses with dynamic (profile) information about the temporal ordering of accesses, and their execution frequency, to create a structure-access database. This database is used to construct a field affinity graph for each structure. These graphs are processed to produce structure field order recommendations. Measurements indicate that field reordering of just 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.

Chapter 7 summarizes the ideas discussed in the dissertation, draws some conclusions, and suggests possible areas for further research.

1.4 Related Work

This section reviews related work on reordering data accesses to improve spatial and temporal locality, and techniques for improving a program's virtual-memory performance.

1.4.1 Improving the Cache Locality of Scientific Programs

Previous research has attacked the processor–memory gap by reordering computation to increase spatial and temporal locality [28, 84, 14]. This work focused on programs with loop nests that access arrays in a regular manner. Gannon *et al.* studied an exhaustive approach that generated all possible permutations of a loop nest [28]. They considered uniformly-generated references, used reference windows to determine the minimum memory locations necessary to maximize reuse within a loop nest, and selected the best permutation based on an evaluation function. Their exhaustive approach is impractical when including loop transformations, such as loop fusion and loop distribution, which combine and create loop nests.

Wolf and Lam developed a loop transformation theory, based on unimodular matrix transformations, that unifies loop transforms like interchange, reversal, and skewing [84]. They introduced the notion of a reuse vector space to capture the potential of optimizing a given loop nest for locality and used this to prune the search space. A heuristic algorithm selects the best combination of loop transformations. Their technique ignores loop bounds even when they are known constants.

Carr et. al used a simple model of spatial and temporal reuse of cache lines to selectively apply compound loop transformations that include loop permutation, reversal, fusion, and distribution [14]. Their model uses reference groups, which are slightly more restrictive than uniformly generated references [28], to calculate reuse.

This work considers an entirely different class of data structures. Pointer-based structures do not support random access, and hence changing a program's access pattern to improve reference locality is impossible in general.

1.4.2 Page-Conscious Data Placement

Database researchers long ago faced a similar performance gap between main memory and disk. They designed specialized data structures, such as B-trees [7, 21], to bridge this gap. In addition, databases use clustering [6, 77, 26, 9] and compression [21] to improve virtual memory performance.

Clustering has also been used to improve virtual memory performance of Smalltalk and LISP systems [52, 71, 82, 42, 23] by reorganizing data structures during garbage collection. Researchers investigated two approaches to using a garbage collector to improve paging behavior of Smalltalk and LISP systems. Static regrouping uses the topology of heap data structures to rearrange structurally-related objects [52, 82], while dynamic regrouping clusters objects according to a program's data access pattern [23]. Moon found that depth-first copying generally yields better virtual memory performance than breadth-first copying for LISP, because it is more likely to place parents and offsprings on the same page, particularly if data structures tend to be shallow, but wide [52]. Wilson *et al.* treated hash tables, which group data in a pseudo-random order, specially, and 'normal' data structures were copied in depth-first order [82]. Their results showed a significant reduction in the incidence of page faults. However, in a later study, the authors found that the optimal grouping of data structure elements was very dependent on the shape and type of the structure being copied [42]. While hierarchical decomposition performed well for trees, it was disappointing for other structures. Court's dynamic regrouping technique takes advantage of specialized hardware to provide incremental garbage collection, which tends to copy objects in program access order, and this can dramatically reduce the number of page faults [23].

More recently, Seidl and Zorn combined profiling with a variety of different information sources present at the time of object allocation to predict the object's reference frequency and lifetime [64]. They showed that program references to heap objects are highly predictable.

These studies focused on a program's paging behavior, not its cache behavior. Our work differs, not only because of the different costs for a cache miss and a page fault, but also because cache blocks are far smaller than memory pages.

In the face of the lack of direct mathematical demonstration, one must be careful and thorough to make sure of the point, ... Nevertheless, a very great deal more truth can become known than can be proven.
—Richard Feynman, Nobel Prize Address

Chapter 2

Cache-Conscious Structure Design

A growing processor–memory performance gap, coupled with the presence of a fast cache memory (or multiple levels of cache memory) between the processor and main memory, divides memory references into two categories — those that hit in the cache with a low access cost, and those that miss in the cache with access costs orders of magnitude higher. Given this memory organization, data structures designed and organized such that most structure references hit in the cache have the potential to significantly improve performance. This chapter explores the design of such cache-conscious data structures.

Pointer structures possess an extremely powerful property of *locational transparency* that facilitates cache-conscious structure design. Elements in a structure can be placed at different memory (and cache) locations without changing a program's semantics. Careful placement of structure elements provides a mechanism to improve the cache locality of pointer-manipulating programs and, consequently, their performance. This chapter investigates three placement design principles—*clustering*, *coloring,* and *compression*—that improve cache performance on uniprocessor systems by increasing a data structure's spatial and temporal locality, and by reducing cache conflicts. Clustering places structure elements likely to be accessed contemporaneously in the same cache block. Coloring segregates heavily and infrequently accessed element in non-conflicting cache regions. Compression reduces structure size or separates the active portion of structure elements from infrequently accessed fields. The chapter discusses the application of these design principles to construct cache-conscious trees.

We performed experimental evaluations of these cache-conscious data structures. 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. These results demonstrate that cache-conscious data structures can offer large performance benefits.

While the design principles described are well known, a contribution of this chapter is to collect these ideas and apply them to improve the cache performance of pointer structures. The rest of the chapter is organized as follows. Section 2.1 provides a brief overview of cache architectures. Section 2.2 discusses cache-conscious structure design principles and explores applying them to trees. Section 2.3 evaluates cache-conscious trees with microbenchmarks. Finally, Section 2.4 briefly discusses related work.

2.1 Background: Cache Architectures

This section provides a brief overview of current microprocessor cache architectures. A detailed understanding of cache architectures is essential for designing cache-conscious data structures. Smith's [59] survey paper, and Hennessy and Patterson [33] contain more information.

Figure 2-1 depicts a typical memory system organization that includes a cache. The cache sits between the processor and main memory and intercepts all memory references.

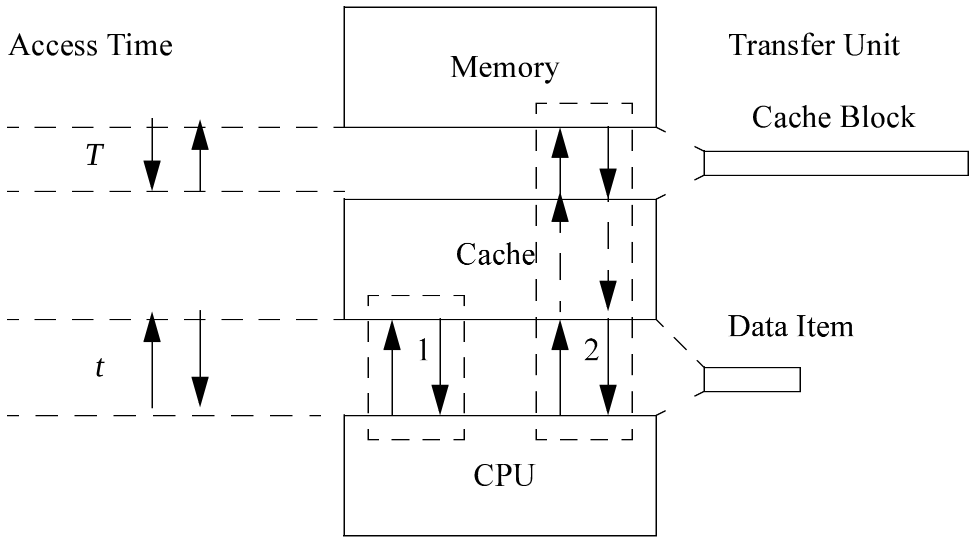


Figure 2-1. Memory system with a cache.

References that hit in the cache (1) incur an access cost of t. References that miss in the cache (2) initiate a cache block transfer to bring the data from main memory into the cache and incur an access cost of $(t+T)$. The average memory-access time is given by

$$\begin{aligned} \textit{Access Time} &= \textit{Hit Time} + (\textit{Miss rate}) \times (\textit{Miss penalty}) \\ &= t + (\textit{Miss rate}) \times T \end{aligned}$$

Cache hierarchies were introduced as a consequence of the growing processor-memory performance gap. The idea is to have a small Level 1 (L1) cache that can match the cycle time of the processor and a large Level 2 (L2) cache to satisfy references that would otherwise access memory. The average memory-access time for such a two-level cache hierarchy is given by

$$\textit{Access Time} = \textit{Hit Time}_{L1} + \textit{Miss rate}_{L1} \times (\textit{Hit Time}_{L2} + (\textit{Miss rate}_{L2} \times \textit{Miss penalty}_{L2}))$$

To amortize the high cost of accessing main memory, data is transferred in units called cache blocks that encompass multiple words. Typical cache block sizes for current microprocessor systems are 16–64 bytes for the L1 cache and 32–256 bytes for the L2 cache. There is a trade-off involved in selecting a cache block size. Larger cache blocks incur a higher miss penalty, but can reduce the number of subsequent misses if the references exhibit spatial locality. However, extremely large cache blocks are undesirable because of the danger of polluting the cache with data that is never referenced. In addition, in multi-processor systems, large cache blocks exacerbate the false sharing problem, in which distinct data items written by different processors happen to reside in the same cache block. This can cause the block to ping-pong between the caches of different processors.

Caches have finite capacity, and require a policy for selecting a block to replace when a cache miss causes a new block to be brought in. Typical L1 cache capacities are 1–128 KB and typical L2 cache capacities are 256 KB–4 MB. If the cache were organized as a fully associative memory, then a block could occupy any position in the cache, and all blocks in the cache would be potential candidates for replacement. However, because fully associative memories are slow, and their implementation costly, caches are either direct-mapped or set-associative. Direct-mapped caches have a fixed cache location for every block of main-memory; thus, many blocks share the same location, and the block to be replaced is completely specified. Direct-mapped caches are simple to build and fast to search, but they tend to have higher miss rates than set-associative caches [34]. Set-associative caches are organized into sets, each of which can contain any number of blocks up to a fixed limit (typically two or four). Every block of main memory is mapped to a set, which occupies a fixed position in the cache. However, there is no restriction on the placement of blocks within a set. Thus set-associative caches represent a compromise between direct-mapped and fully-associative caches wherein every block within a set is a candidate for replacement. The cache replacement policy (typically LRU) selects one of these blocks.

Caches also differ in the mechanism used to access a cache block. They can be virtually-indexed [8] or physically-indexed. Virtually-indexed caches use the virtual address to access the cache as opposed to physically-indexed caches which use the physical address. The advantage of using the virtual address is that cache hits are faster since they do not have to wait for the TLB to translate the virtual address to a physical one prior to cache lookup. However, virtually-indexed caches have a problem with synonyms that arises when the operating system uses the same virtual address for two different physical addresses. Typically, L1 caches are often virtually-indexed, and L2 caches are physically-indexed.

A more recent innovation in cache design is a lock-up free or non-blocking cache [41]. A non-blocking cache allows for multiple, concurrent outstanding misses. These are beneficial in current microprocessors that support out-of-order execution [75], because instructions that do not require the data that caused the cache miss can continue to execute and access the cache. Non-blocking caches can also reduce the effective memory-access time by overlapping multiple misses. On the other hand, non-blocking caches have complex hardware requirements and for many programs it is unclear whether their benefits justify their cost.

2.2 Cache-Conscious Structure Design

This section discusses cache parameters that are relevant to cache-conscious structure design and explores three placement principles—*clustering*, *coloring*, and *compression*. The example in this discussion is binary trees.

2.2.1 Cache Parameters

Caches can be parameterized by capacity, block size, associativity, indexing (virtual or physical), write policy, and non-blocking degree. This thesis 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 thesis). Write buffers do a good job of hiding write latency, hence write hit/miss policy is unlikely to have a large impact on performance. No such technique exists for read latency, which is the primary problem. In addition, while many architectures have physically-indexed L2 caches, operating systems such as Microsoft Windows NT, Sun Solaris, and SGI IRIX have a deterministic page-mapping policy that maps consecutive virtual pages to non-conflicting physical pages [38]. Conflicts only occur between pages whose virtual addresses differ by a multiple of the cache set size.

A *cache configuration C* can be expressed as a triple < *c, b, a* > where:

c = cache capacity in sets

b = cache block size in bytes

a = cache associativity.

For example, a two-way set-associative 64 Kbyte cache with 64 byte blocks is <512, 64, 2>.

2.2.2 Cache-Conscious Design Principles

This section discusses three general data placement design principles—*clustering, coloring,* and *compression*—that can be combined in a wide variety of ways to produce cache-efficient data structures. Clustering attempts to pack data structure elements likely to be accessed contemporaneously into a cache block. Clustering improves spatial and temporal locality and provides implicit prefetching. Figure 2-4 illustrates clustering applied to a binary tree.

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 2-2 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

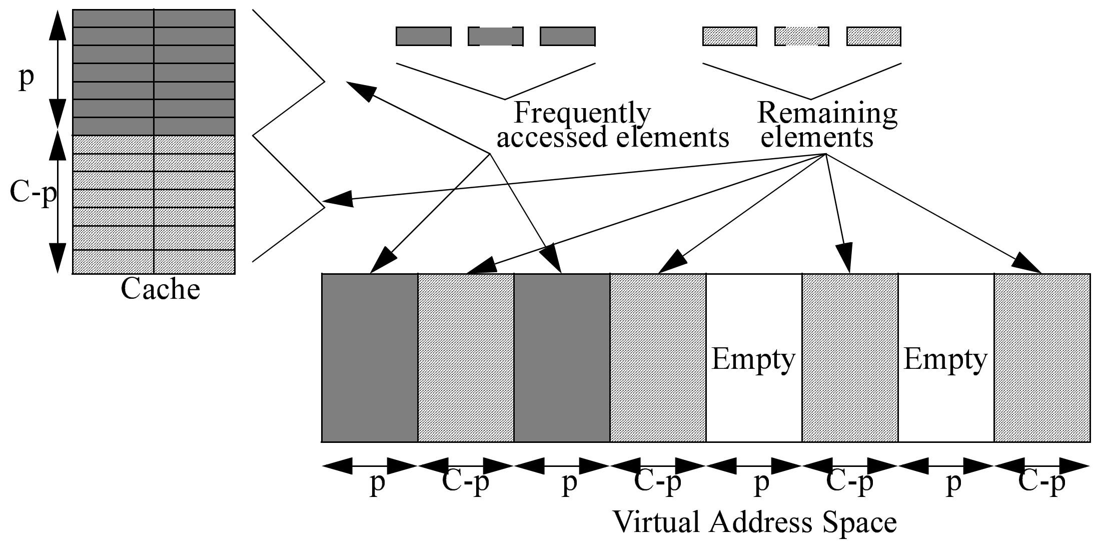
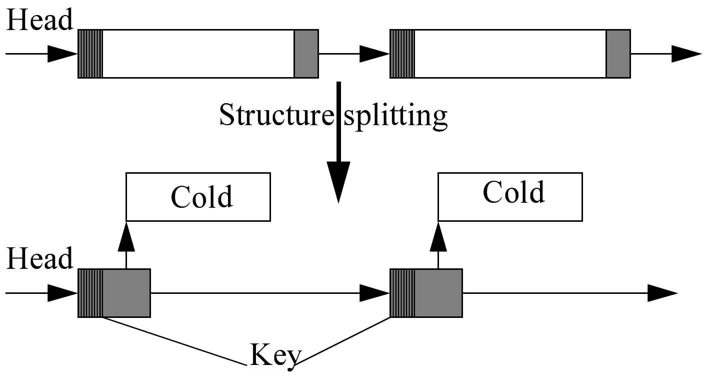


Figure 2-2. Coloring data structure elements to reduce cache conflicts.

region (i.e., such that they don't 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 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.

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 [21], 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



```
for (p = Head; p != NULL; p = p->next)
{
   if(p->key == Key)
      break;
}
if (p != NULL)
   Examine other fields of p;
```

Figure 2-3. Compression through structure splitting.

stored at known offsets in an array. Another example is the tree structure shown in Figure 2-5, which eliminates the internal subtree pointers from the clusters in the tree shown in Figure 2-4.

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 does not compress the data structure. Instead, it separates heavily accessed (hot) portions of data structure elements from rarely accessed (cold) portions (Figure 2-3). The heavily accessed portions can then be clustered to improve locality.

2.2.3 Cache-Conscious Trees

An effective way to cluster a tree is to pack subtrees[1] into a cache block. Figure 2-4 illustrates subtree clustering for a binary tree. An intuitive justification for binary subtree clustering is as follows (a detailed analysis is given in Section 3.3). For a series of random

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

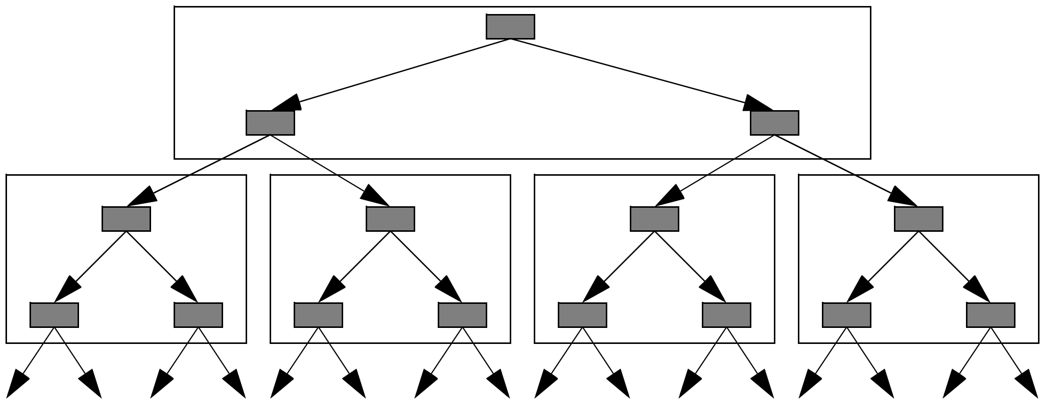


Figure 2-4. Subtree clustering.

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. In this case, the expected number of accesses to the block is:

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

Of course, this analysis assumes a random access pattern. For specific access patterns, such as depth-first search, other clustering schemes may be better. 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.

For a tree, the most heavily accessed elements are the nodes near the root of the tree. Hence, the tree can be colored as shown in Figure 2-2 with the top levels of the tree marked as frequently accessed elements. These elements are mapped to a unique portion of the cache that does not conflict with the rest of the tree.

After a tree is clustered into subtrees (see Figure 2-4), the structure can be compressed by eliminating pointers between nodes in the same cache block subtree, as shown in Figure 2-5. Further compression, by applying structure splitting to the tree shown in

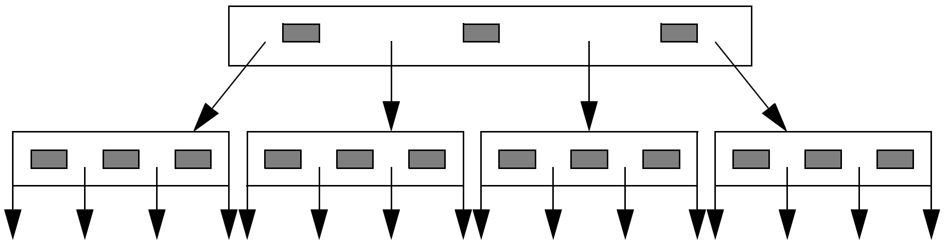


Figure 2-5. Compression by pointer elimination.

Figure 2-5 results in an object that is structurally similar to a B-tree (except for the space in a B-tree node that is reserved for insertions).

The previous discussion showed that clustering and compressing a binary tree produces a structure similar to a B-tree. Another approach to constructing a cache-conscious tree is to start with a B-tree, which was originally designed as an on-disk data structure that addresses the performance gap between memory and disk, and adapt it to an in-core structure that addresses the processor-memory performance gap. Due to the high cost of transferring data between disk and memory, B-tree 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 2-6).

The graph has a cost minimum when the in-core B-tree nodes are 96 bytes, although the cache block size is only 64 bytes. This seemingly surprising result makes sense 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

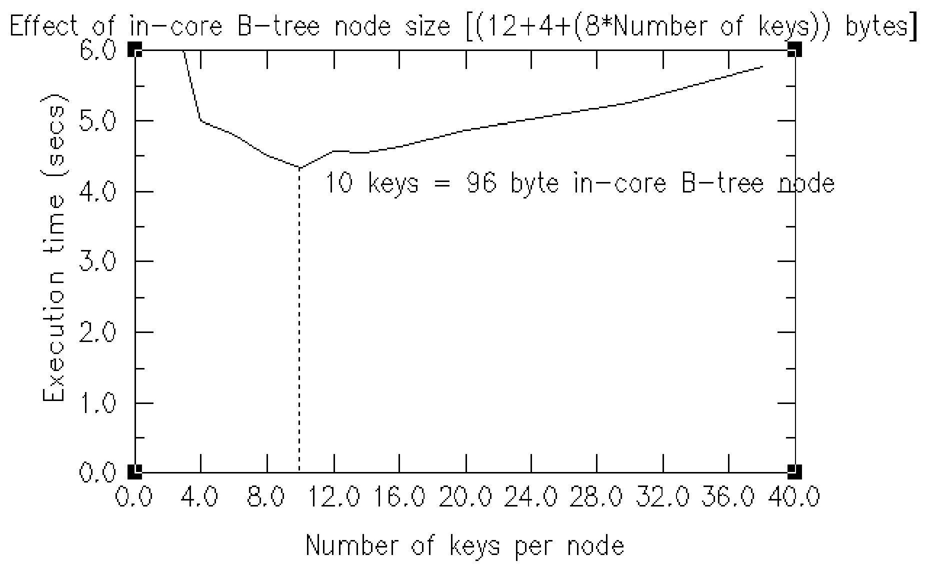


Figure 2-6. B-tree node size.

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

Section 2.3.2 compares the performance of this in-core B-tree (with the top tree levels colored to reduce cache conflicts), with the subtree clustered tree discussed earlier.

2.3 Evaluation of Cache-Conscious Structure Design

This section evaluates cache-conscious trees with microbenchmarks.

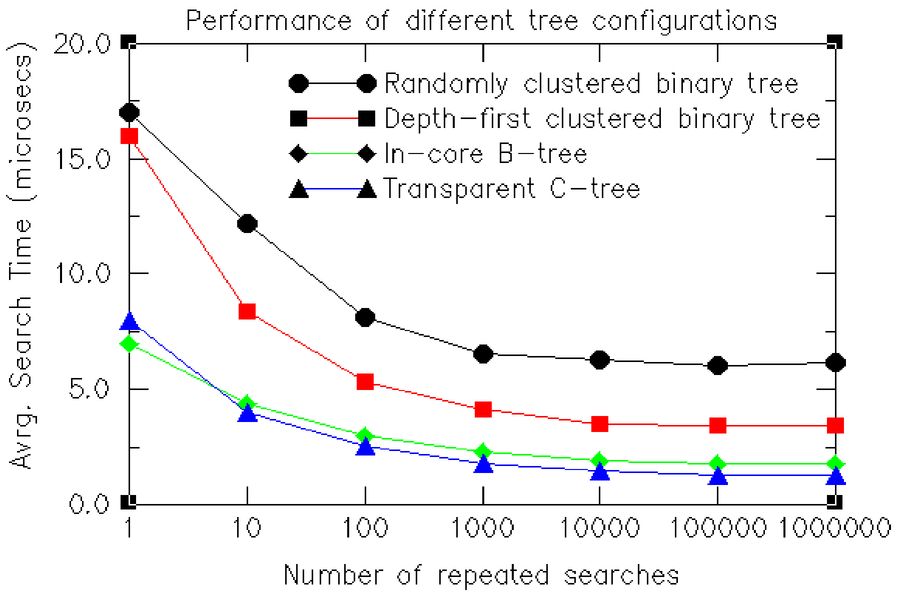


Figure 2-7. Binary tree microbenchmark.

2.3.1 Methodology

We ran the benchmarks on a single processor of 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 16KB direct-mapped L1 data cache with 16 byte lines, and a 1 MB direct-mapped L2 cache with 64 byte lines. An L1 data cache hit takes 1 cycle (i.e., $t_h = 1$). An L1 data cache miss, with an L2 cache hit, costs 6 additional cycles (i.e., $t_{mL1} = 6$). An 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.

2.3.2 Tree Microbenchmark

The tree microbenchmark measures the performance of the cache-conscious binary search tree described in Section 2.2.3 without compression applied, a data structure we call a transparent C-tree. We compared its performance with an in-core B-tree, also colored to reduce cache conflicts, and with random and depth-first clustered binary trees. The microbenchmark does not perform insertions or deletions. The tree contained 2,097,151 keys and consumed 40 MB of memory (forty times the size of the L2 cache). 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 2-7 shows that both B-trees and transparent C-trees outperform randomly clustered binary trees by up to a factor of 5, and depth-first clustered binary trees by up to a factor of 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 in-core B-trees to perform better than transparent C-trees when trees change due to insertions and deletions.

2.4 Related Work

Database researchers long ago faced a similar performance gap between main memory and disk speeds. They designed specialized data structures, such as B-trees, to bridge this gap [7, 21]. In addition, databases use clustering [6, 77, 26, 9] and compression [21] to improve virtual memory performance. Section 2.2.3 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 by reorganizing data structures during garbage collection [52, 71, 82, 42, 23]. However, 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.

The test of science is its ability to predict.
Richard Feynman, The Feynman Lectures, book II

Chapter 3

A Framework for Cache-Conscious Structure Design

Although the cache-conscious data placement design principles discussed in the previous chapter can improve a structure's spatial and temporal locality, their description is ad hoc. The framework presented in this chapter addresses this difficulty by quantifying their performance advantage. The framework permits *a priori* estimation of the benefits of these design principles. 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. In addition, it provides intuition for understanding the impact of data layout on cache performance.

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. Some examples are: searching for an element in a tree, or traversing a linked list. To make the details concrete, this chapter applies the analytic framework to predict both the steady-state performance and the start-up or transient performance of cache-conscious trees. In addition, it reports on experiments to validate the model's predictions.

The rest of the chapter is organized as follows. Section 3.1 presents our analytic framework for characterizing the performance benefits of cache-conscious pointer structures. Section 3.2 applies the model to analyze the steady-state performance of cache-conscious trees. Section 3.3 analyzes their transient or start-up behavior. Section 3.4 discusses experiments to validate the model's predictions via microbenchmarks. The chapter concludes with a brief discussion of related work.

3.1 Analytic Framework

This section discusses our analytic framework for characterizing the performance of cache-conscious pointer structures.

3.1.1 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

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

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 the expected value of this memory access expression. 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, denoted by $m_a(p)$, as

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

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, denoted by m_s, as

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

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 bytes, 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 complete, balanced binary search tree of *n* elements. 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 *K* represent the average number of structure elements residing in the same cache block that are required for the current pointer-path access. *K* is a measure of a data structure's spatial locality for the access function *D*. From the definition of *K* it follows that

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

For the *i-th* pointer-path access, let *R(i)* 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(i)* is the number of elements that are reused during the *i*-th pointer-path access, and is a measure of a data structure's temporal locality. From the definition of *R(i)* it follows that

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

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

m(i) = (number of cache misses) / (total references)

$$m(i) \cong \frac{\frac{D \text{ Ð } R(i)}{K}}{D} = \frac{1 \text{ Ð } \frac{R(i)}{D}}{K}$$

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 2.2.2), then *R(i)* will approach a constant value R_s when this steady state is reached. Since *D* and *K* are both independent of *i*, for a large, random sequence of pointer-path accesses *p,* all of the same type, the amortized steady-state miss rate m_s of a data structure can be approximated by its amortized miss rate $m_a(p)$ as follows

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

This equation can be used to analyze the steady-state behavior of a pointer-based data structure, and the previous equation to analyze its transient start-up behavior.

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

Figure 3-1. Cache-conscious speedup.

3.1.2 Speedup Analysis

The cache performance model given in the previous section 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*. *K* can be increased by intelligently *clustering* data structure elements into cache blocks. *K* can also be increased by *compressing* data structure elements, which also permits greater 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 in terms of cache miss rates for the speedup that results from applying cache-conscious 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.

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

When only the structure layout is changed, the number of memory references remains the same and the equation reduces to that in Figure 3-1.

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

Figure 3-2. Best-case cache-conscious speedup.

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 $(m_{L1})_{Naive} = (m_{L2})_{Naive} = 1$. Thus, the best-case cache-conscious speedup reduces to that in Figure 3-2.

3.2 Model Application: Steady-State Analysis

This section demonstrates how to calculate the steady-state performance of a cache-conscious tree (see Section 2.2.3) subjected to a series of random key searches.

3.2.1 Cache-Conscious Trees

Consider a complete, balanced 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 that 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 the next $c/2$ sets of the cache (other divisions of the cache are possible).

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. If the number of tree searches is

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

Figure 3-3. Cache-conscious binary tree.

large, we can ignore the start-up behavior and approximate the data structure's performance by its amortized steady-state miss rate.

From the above discussion, we have $K = log_2(k+1)$ and $R_s = \mathrm{l}og_2(c/2 \times k \times a + 1)$. That is, 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. Applying the steady-state miss rate equation, we get the result shown in Figure 3-3.

3.3 Model Application: Transient Analysis

In this section, we demonstrate how amortized analysis [73] can be applied to our cache model to compute the transient or start-up behavior of cache-conscious trees.

Amortized analysis, which averages the cost of a sequence of operations, is a powerful technique for the complexity analysis of data structures. In many cases, a worst-case analysis, in which the worst-case times of individual operations is summed, is overly pessimistic, as it ignores correlated effects of the operations on the data structure. In such a situation, an amortized analysis, in which the running time per operation over a (worst-case) sequence of operations is averaged, can yield an answer that is more realistic.

The combination of a sequence of accesses to a data structure and the presence of a cache results in correlation between different accesses. We adapt amortized analysis to this situation and use the aggregate method of amortized analysis to compute worst-case bounds for the cache-performance parameters, K and R. Using the aggregate method, we show that a sequence of pointer-path accesses σ, has a total worst-case cost T_σ, for any σ. Thus, in the worst case, the amortized cost per pointer-path access is $T_\sigma/|\sigma|$.

3.3.1 Cache-Conscious Trees

To demonstrate this technique, we apply the model to analyze the cache performance of a series of searches on a complete binary tree. To simplify matters, assume the cache configuration to be a fully-associative, blocking cache of infinite capacity, where the cache block size is b words. For such a cache configuration, the cost of a binary tree search can be reduced by intelligently clustering tree nodes in cache blocks.

Consider a complete binary tree of n nodes. Let the size of a node be e words. If the cache block size is b words, up to $\lfloor b/e \rfloor$ nodes of the tree can be clustered in a cache block. To simplify calculations, let us assume that $k = \lfloor b/e \rfloor$ is of the form $2^r - 1$. This allows complete subtrees of the appropriate size to fit in a cache block. Let us additionally assume that $log_2(k+1)$ is a factor of $log_2(n+1)$. This permits a division of the complete binary tree into complete binary subtrees. We cluster the binary tree into subtrees and place these in cache blocks. To do this we recursively divide the tree into subtrees containing $\lfloor b/e \rfloor$ nodes, starting at the root, as shown in Figure 2-4. All the assumptions made here are solely to simplify the calculations, and do not affect the generality of the analysis.

Now let us consider a series of i worst-case searches on this subtree clustered binary tree, where $i < (n+1) / (k+1)$. To simplify matters, we assume that i is a power of $(k+1)$. We use the term worst-case search to imply that each successive search has minimal correlation with elements accessed during the previous searches. Since the cache configuration is fully-associative and has infinite capacity, this greedy worst-case approach is in fact the worst-case when all sequences are considered. The first search accesses $log_2(n+1)$ elements, all of which miss in the cache, and this corresponds to $log_{(k+1)}(n+1)$ cache block misses. The second "worst-case" search accesses $log_2(n+1)$ elements, all of which miss in the cache, except for the first $log_2(k+1)$ elements, which were brought in by the previous access, resulting in $log_{(k+1)}(n+1)$ – 1 cache block misses, and so on for each succeeding access. Thus we have

$$\begin{array}{rcl}
\underline{\text{Pointer Path Access}} & & \underline{\text{Cache Misses}} \\
1 & \rightarrow & \log_{(k+1)}(n+1) \\
2\ldots(k+1) & \rightarrow & \log_{(k+1)}(n+1)\text{Đ}\,1 \\
(k+1)+1\ldots(k+1)^2 & \rightarrow & \log_{(k+1)}(n+1)\,\text{Đ}\,2 \\
\ldots & \rightarrow & \ldots \\
(k+1)^{\log_{(k+1)}i\,\text{Đ}\,1}+1\ldots i & \rightarrow & \log_{(k+1)}(n+1)\,\text{Đ}\,\log_{(k+1)}i
\end{array}$$

Summing up the l.h.s. and r.h.s., we have

$$\begin{array}{rcl}
\underline{\text{Total Number of Connected Accesses}} & & \underline{\text{Total Number of Cache Misses}} \\
i & \rightarrow & i\log_{(k+1)}(n+1)\,\text{Đ}\,T
\end{array}$$

$$T = k(1+2\times(k+1)+\ldots+\log_{(k+1)}i\times(k+1)^{\log_{(k+1)}(i)\,\text{Đ}\,1})$$

Now the series S_n has the closed form

$$S_n = 1+2x+3x^2+\ldots+nx^{n\,\text{Đ}\,1} = \frac{nx^{n+1}\,\text{Đ}\,(n+1)x^n+1}{(x\,\text{Đ}\,1)^2}$$

Using this to evaluate *T*, with *(k+1)* replacing *x* and l*og*$_{(k+1)}$*i* replacing *n*, we have

$$T = i\times\log_{(k+1)}i\,\text{Đ}\,\frac{(i\,\text{Đ}\,1)}{k}$$

$$T = k\left(\frac{\log_{(k+1)}i\times i\times(k+1)\,\text{Đ}\,(\log_{(k+1)}i+1)\times i+1}{k^2}\right)$$

Back-substituting for *T* and calculating the amortized number of cache misses per connected access (i.e., cache misses / i), we have

$$1 \rightarrow \log_{(k+1)}(n+1)\ \text{Ð}\ \log_{(k+1)} i + \frac{(i\ \text{Ð}\ 1)}{i \times k}$$

The last term in this is negligible, and thus the expression can be rewritten as

$$1 \rightarrow O\left(\frac{\log_2(n+1)\ \text{Ð}\ \log_2 i}{\log_2(k+1)}\right)$$

Now for a binary tree search, $D = log_2(n+1)$ for a single pointer path access. Then

$$Miss\ \text{Ð}\ Rate\ =\ \frac{CacheMisses}{References}\ =\ O\left(\frac{1\ \text{Ð}\ \dfrac{\log_2 i}{\log_2(n+1)}}{\log_2(k+1)}\right)$$

Comparing this with the transient miss rate equation, we have

$$K\ =\ O(\log_2(k+1))$$

$$R(i)\ =\ O(\log_2 i)$$

Similar analysis can be performed for alternative tree layouts or designs.

3.4 Model Validation

This section addresses the question of how well this model predicts performance. Two aspects are addressed. First, we show that the individual techniques of cache-conscious design and layout correspond to the model. Second, we show that 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.

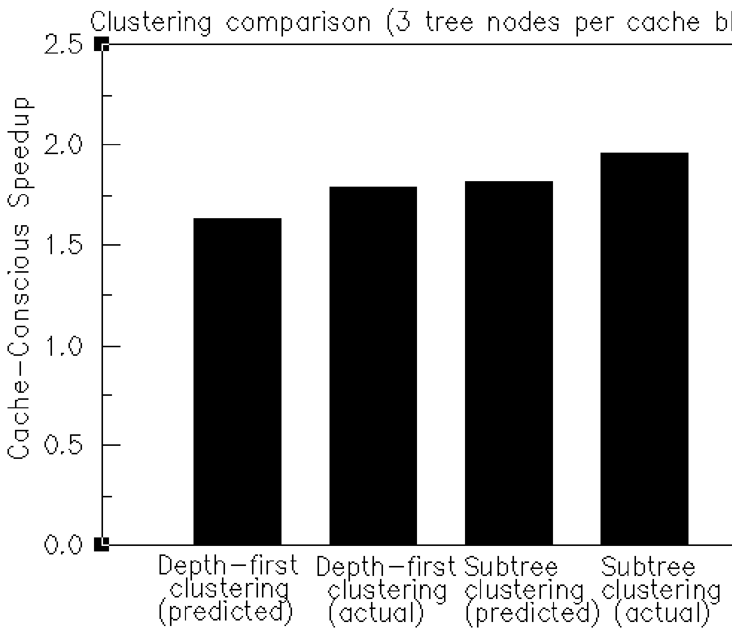


Figure 3-4. Predicted and actual effects of clustering.

The experimental setup is the same as before (see Section 2.3.1). 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.

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 the equation for best-case cache-conscious speedup (see Figure 3-2), we obtained the predictions shown in Figure 3-4. The model underestimates the speedup for both clustering techniques by only 8–9%.

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

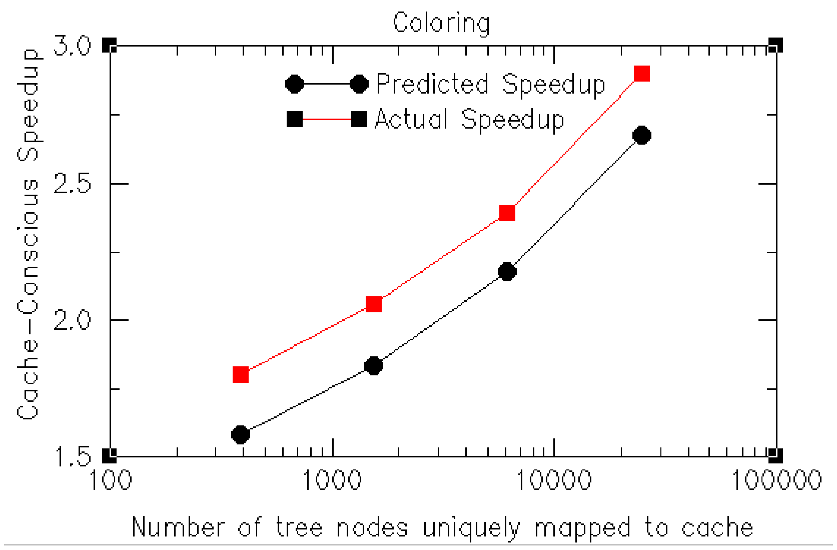


Figure 3-5. Predicted and actual effects of coloring.

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 3-5 shows that the model underestimates the improvement by only 8–14% and accurately predicts the shape of the speedup curve.

Finally, we applied 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 3-6. As the graph shows, the model underestimated the actual speedup by only 15% and accurately predicted the shape of the curve.

3.5 Related Work

Several other cost models have tried to capture the hierarchical nature of memory systems. The Uniform Memory Hierarchy (UMH) model of Alpern *et al.* models memory as

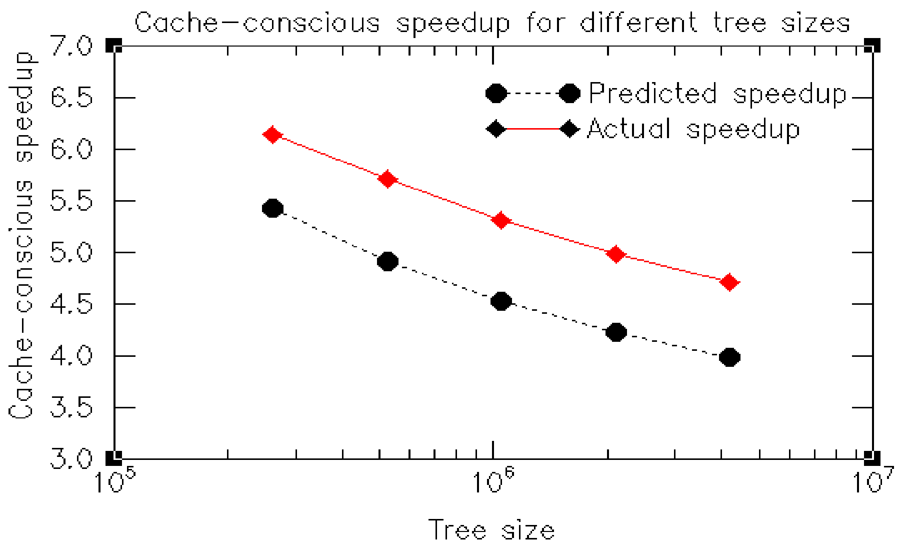


Figure 3-6. Predicted and actual speedup for C-trees.

a sequence of increasingly large modules <M_0, M_1,... >, in which each module M_u, is represented with 3 parameters, <s_u, n_u, l_u> [5]. 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, but addresses non-oblivious computations.

Researchers have also used empirical models of program behavior [4, 63, 70] to analyze cache performance [59, 66, 31]. 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 [65]. 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, reflected in the 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 workload behavior. 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 proposed the simple 3C model, which classifies cache misses into three categories—compulsory, capacity, and conflict [29]. The model provides an intuitive explanation for the causes of cache misses, but it lacks predictive power. These models focus on analyzing and predicting a program's cache performance, while we focus on the cache performance with respect to (sequences of) operations on individual in-core pointer structures.

Lam et al. [43] 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 a sequence of pointer-path accesses to an in-core pointer-based data structure.

LaMarca and Ladner explored the interaction of caches and sorting algorithms [44, 45]. In addition, they constructed a cache-conscious heap structure that clustered and aligned heap elements. Their "collective analysis" models an algorithm's behavior for direct-mapped caches and obtains accurate predictions. Their framework relies on the "independence reference assumption" [4], and is algorithm-centric, whereas ours is data structure-centric, and specifically targets correlations between multiple accesses to the same data structure.

Nothing of him that doth fade
But doth suffer a sea-change
Into something rich and strange.
—Shakespeare, The Tempest

Chapter 4

Cache-Conscious Structure Layout

As the previous chapter showed, careful design of data structures provides a mechanism to improve the cache locality of pointer manipulating programs, and consequently their performance. However, cache-conscious data structure design requires detailed knowledge of a program's code and data structures, familiarity with the architecture of the target machine, and considerable programmer effort.

These costs may limit the use of cache-conscious data structures to performance critical portions of code written by expert programmers, much like assembly programming is used today. To make the performance benefits of cache-conscious structures available to the average programmer, we explore two strategies—*cache-conscious reorganization* and *cache-conscious allocation*—for facilitating the creation of cache-conscious pointer structure layouts. In addition, to make these discussions concrete and demonstrate the feasibility of this approach, we describe two semi-automatic tools—`ccmorph` and `ccmalloc`—that embody these strategies. Measurements show that the cache-conscious data layouts produced by `ccmorph` and `ccmalloc` produce large performance benefits.

Cache-conscious reorganization can utilize structure topology or profiles of data-access patterns to transform pointer structure layouts. This chapter discusses the use of structure topology to produce cache-conscious structure layouts. The next chapter explores the use of program profiles for this purpose. The topology approach to structure reorganization is incorporated in a utility—`ccmorph`—that reorganizes tree-like structures, such as trees, lists, and chained hash tables, by clustering and coloring the structure as described in Chapter 2.

Cache-conscious allocation improves conventional heap allocation by attempting to co-locate contemporaneously accessed data elements in the same physical cache block. The chapter discusses `ccmalloc`, a memory allocator that implements this strategy.

Both of these tools require little effort and understanding on the part of a programmer. To use `ccmorph`, a programmer need only supply a function used to traverse the data structure. In the case of `ccmalloc`, a programmer must only specify an additional argument to each call to `malloc`—a pointer to a structure element likely to be in use contemporaneously with the one to be allocated.

Our experimental evaluations demonstrate the performance benefits of these approaches to cache-conscious data placement. For some pointer-intensive programs in the Olden benchmark suite [60], semi-automatic cache-conscious data placement improves performance by 28–194%, and even outperformed state-of-the-art software prefetching by 3%–194%. We also applied the techniques to full application programs: RADIANCE [80], a widely used ray-tracing program, ran 42% faster, and VIS [10], a model-verification package, improved by 27%. Significantly, applying `ccmalloc` to the 160,000 line VIS code required little understanding of the application, and took only a few hours.

The rest of the chapter is organized as follows. Section 4.1 explores cache-conscious data reorganization and describes `ccmorph`. Section 4.2 investigates cache-conscious heap allocation and discusses `ccmalloc`. Section 4.3 evaluates the performance benefits of our approaches to producing cache-conscious structure layouts. Finally, Section 4.4 briefly discusses related work.

4.1 Cache-Conscious Data Reorganization

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 data reorganization addresses this problem by changing a structure's layout to correspond to its access pattern. General graph-like structures require a detailed profile of a program's data access patterns for successful data reorganization [12, 20]. However, a very important class of structures (trees) possess topological properties that permit cache-conscious data reorganization without profiling. This section presents a transparent (semantics-preserving) cache-conscious tree reorganizer (`ccmorph`) that applies the clustering and coloring techniques described in Chapter 2.

Reorganization 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.

4.1.1 ccmorph

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 [20]). 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 2.2.3.

```
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);
   }
}
```

Figure 4-1. `ccmorph`: Transparent cache-conscious data reorganization.

`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, and cache parameters. For example, Figure 4-1 contains the code used to reorganize the quadtree data structure in the Olden benchmark *perimeter.* The programmer supplies the *next_node* function.

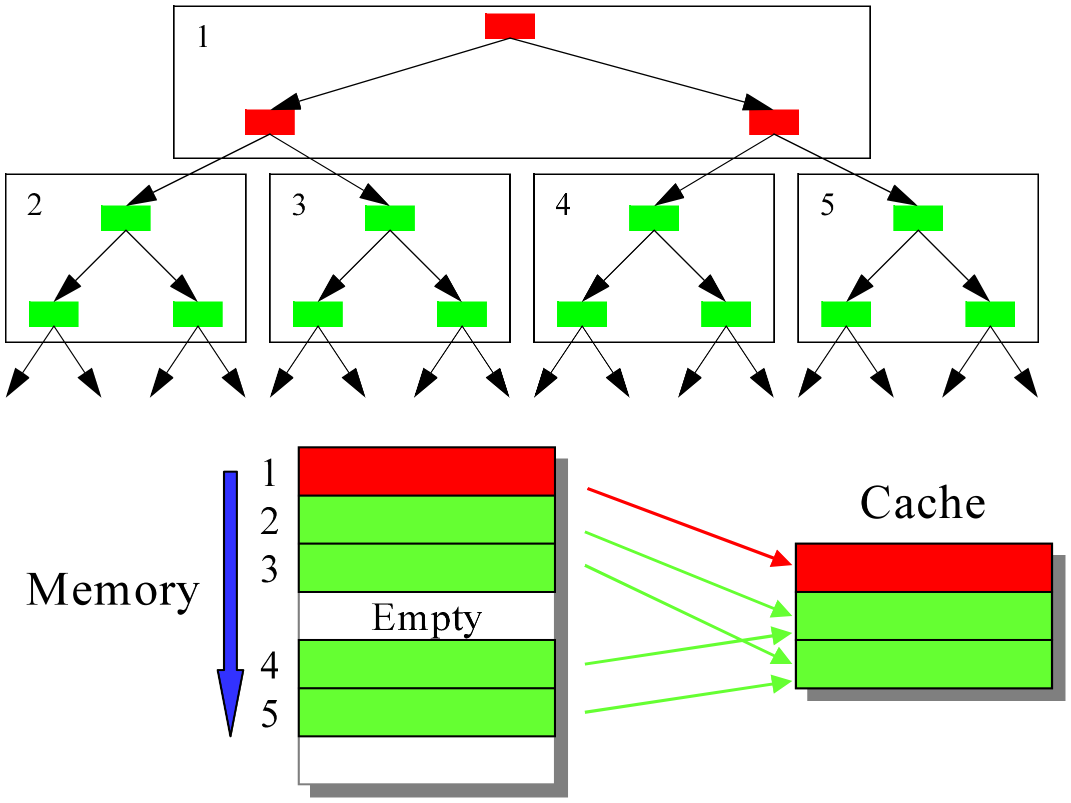


Figure 4-2. Cache-conscious tree reorganization.

`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 4-2). 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.

The effectiveness of `ccmorph` is discussed in Section 4.3.

4.2 Cache-Conscious Heap Allocation

Although `ccmorph` requires little programming effort, it currently only works for tree-like structures whose elements can be moved. In addition, incorrect usage of `ccmorph` can affect program correctness. A complementary approach, which also requires little programming, is to perform cache-conscious data placement when elements are allocated. In general, a heap allocator is invoked many more times than a data reorganizer, so it must use techniques that incur low overhead. Another difference is that data reorganizers operate on entire structures with global techniques, such as coloring, whereas a heap allocator has an inherently local view of the 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.

4.2.1 ccmalloc

`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. Figure 4-3 contains code from the Olden benchmark *health* that illustrates the approach. Our experience with `ccmalloc` indicates that a programmer unfamiliar with an application can sometimes select a suitable parameter by local examination of code surrounding the allocation statement and obtain good results (see Section 4.3).

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 our system (Sun UltraSPARC 1), L1 cache blocks are 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

```
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 4-3. `ccmalloc`: Cache-conscious heap allocation.

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.

An important issue is where to allocate a new data item if a cache block has insufficient space. `ccmalloc` tries to put the new data item as close to the existing item as possible. Putting the items on the same virtual-memory page is likely to reduce the program's working set, thereby improving TLB performance by exploiting the strong hint from the programmer that the two items are likely to be accessed together. Moreover, putting them on the same page ensures they will not conflict in the cache. There are several possible strategies to 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 remainder of the block for future calls on `ccmalloc`. 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.

4.3 Evaluation of Cache-Conscious Data Placement

To evaluate our cache-conscious placement techniques, we use two large, real-world applications. In addition we performed detailed, cycle-by-cycle simulations on four benchmarks from the Olden suite to break down where the time is spent. The macrobenchmarks were a 60,000 line ray-tracing program and a 160,000 line formal-verification system. The Olden benchmarks are a variety of pointer-based applications written in C.

4.3.1 Methodology

We ran the benchmarks on a Sun Ultraserver E5000. The machine configuration is described in Section 2.3.1. All benchmarks were compiled with gcc (version 2.7.1) at the -O2 optimization level and run on a single processor of the E5000.

4.3.2 Macrobenchmarks

We 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 [80]. 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 octree that represents the scene to be modeled. This structure is already 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 `ccmalloc` in this case). We changed the octree to use subtree clustering and colored the data structure to reduce cache conflicts. The performance results include the overhead of restructuring the octree.

VIS (Verification Interacting with Synthesis) is a system for formal verification, synthesis, and simulation of finite-state systems [10]. VIS synthesizes finite-state systems and/or verifies properties of these systems from Verilog descriptions. The fundamental data

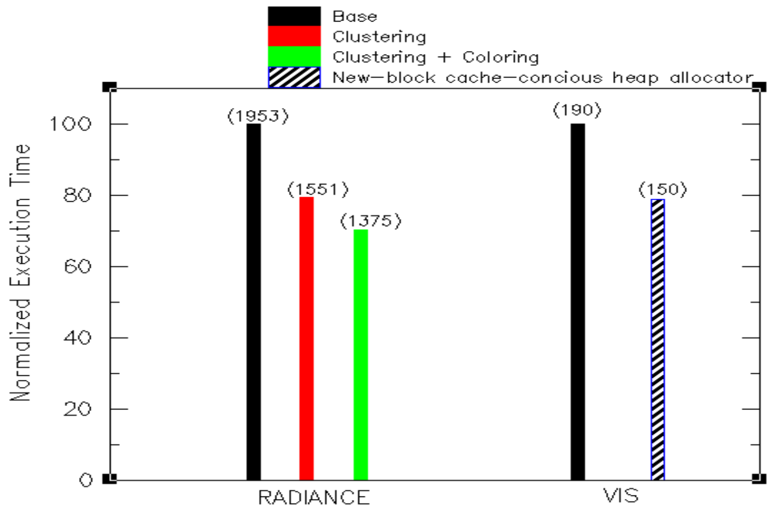


Figure 4-4. RADIANCE and VIS Applications. Actual execution times above each bar.

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, `ccmorph` cannot be used. However, we modified VIS to use our `ccmalloc` allocator with the *new-block* strategy (since it consistently performed well, see Section 4.3.3).

Figure 4-4 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 (less than three hundred lines of code) to these 60–160 thousand line programs produced large performance improvements. In particular, the modifications to VIS were accomplished in a few hours, with little understanding of the application.

4.3.3 Olden Benchmarks

We performed detailed, cycle-by-cycle uniprocessor simulations of the four Olden benchmarks [60] using RSIM [54] to break down where the time is spent. RSIM is an execution-driven simulator that models a dynamically-scheduled, out-of-order processor sim-

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 (# of outstanding misses)	8, 8

Table 4.1: Simulation Parameters.

ilar to the MIPS R10000. Its aggressive memory hierarchy includes a non-blocking, multiported, and pipelined L1 cache, and a non-blocking and pipelined L2 cache. Table 4.1 contains the simulation parameters.

Table 4.2 describes the four Olden benchmarks. We used the RSIM simulator to perform a detailed comparison of our semi-automated cache-conscious data placement implementations—`ccmorph` (*clustering only*, *clustering and coloring*), and `ccmalloc` (*closest*, *first-fit*, and *new-block* strategies)—against other latency reducing schemes, such as hard-

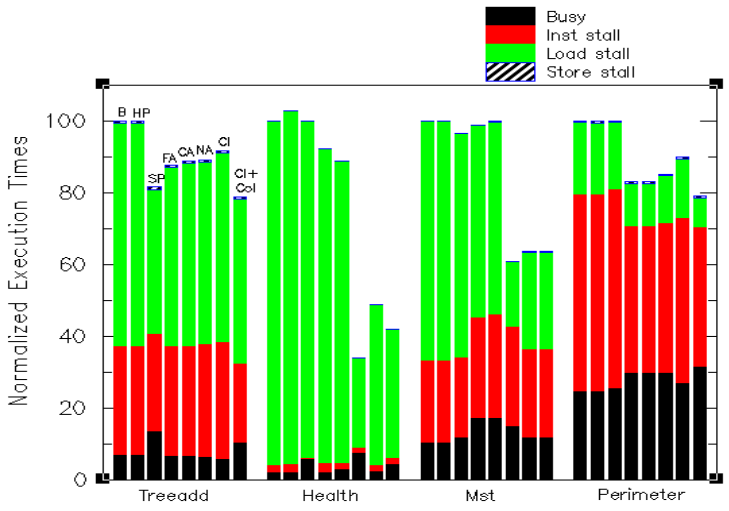
Name	Description	Main Pointer-Structures	Input Data Set	Memory Allocated
Tree-Add	Sums the values stored in tree nodes	Binary tree	256 K nodes	4 MB
Health	Simulation of Colombian health care system	Doubly linked lists	max. level = 3, max. time =3000	828 KB
Mst	Computes minimum spanning tree of a graph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

Table 4.2: Benchmark characteristics.

ware prefetching (prefetching all loads and stores currently in the reorder buffer) and software prefetching (we implement Luk and Mowry's greedy prefetching scheme [50] by hand).

Figure 4-5 shows the results. Execution times are normalized against the original, unoptimized code. We used a commonly applied approach to attribute execution delays to various causes [55, 61]. 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 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 dominant traversal order, which produces a "natural" cache-conscious layout. However, all cache-conscious data placement implementations outperform hardware prefetching, are competitive with software prefetching for *treeadd*, and outperform both software and hardware prefetching for *perimeter*. The `ccmalloc`-*new-block* allocation policy requires 12% and 30% more memory than *closest* and *first-fit* allocation policies, for *tree-*



B - Base
HP - H/W prefetch
SP - S/W prefetch
ccmalloc:
FA - First-fit allocator
CA - Closest allocator
NA - New-block allocator
ccmorph:
Cl - Clustering only
Cl+Col - Clustering and coloring

Figure 4-5. Performance of cache-conscious data placement.

add and *perimeter*, respectively (primarily due to leaf nodes being allocated in new cache blocks).

Health's primary data structures are linked lists, to which elements are repeatedly added and removed. The cache-conscious version periodically invoked `ccmorph` to reorganize the lists (no attempt was made to determine the optimal interval between invocations). Despite this overhead, `ccmorph` significantly outperformed both software and hardware prefetching. Not surprisingly, the `ccmalloc`-*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 collision resolution. It constructs this structure at program start-up and it does not change during program execution. As for *health*, the `ccmalloc`-*new-block* allocator and `ccmorph`, significantly outperformed other schemes. `ccmorph`'s coloring did not have much impact since the lists were short. However, with short lists and no locality between lists, incorrect placement incurs a high penalty. The `ccmalloc`-*new-block* allocator significantly outperformed both *first-fit* and *closest* allocation schemes at a cost of only 3% extra memory.

Technique	Data Structures	Program Knowledge	Architectural Knowledge	Source Code Modification	Performance
CC Design	Universal	High	High	Large	High
`ccmorph`	Tree-like	Moderate	Low	Small	Moderate–High
`ccmalloc`	Universal	Low	None	Small	Moderate–High

Table 4.3: Summary of cache-conscious data placement techniques.

In summary, `ccmorph` 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 `ccmalloc`-*new-block* allocation strategy alone produced speedups of 20–194% over prefetching. In addition, the `ccmalloc`-*new-block* allocator compares favorably with the other allocations schemes, with low memory overhead (with the exception of *perimeter*). To confirm that this performance improvement is not merely an artifact of our `ccmalloc` implementation, we ran a control experiment where we replaced all `ccmalloc` parameters by null pointers. The resulting programs performed 2%–6% worse than the base versions that use the system `malloc`.

4.3.4 Discussion

Table 4.3 summarizes the trade-offs among the cache-conscious data placement techniques. While incorrect use of `ccmorph` can affect program correctness, misapplying `ccmalloc` will only affect program performance. In addition, the techniques in this chapter focus primarily on single data structures, though `ccmalloc` can co-locate elements from different structures. Real programs, of course, use multiple data structures, though often references to one structure predominate. Our techniques can be applied to each structure in turn to improve its performance. The next chapter considers interactions among different structures.

Our cache-conscious structure layout techniques place contemporaneously accessed elements in the same cache block. While this will always improve cache performance on uniprocessors, for multiprocessor systems it depends on whether the data items are accessed by same processor or by different processors. In the latter case, co-locating the data elements could exacerbate false-sharing.

4.4 Related Work

Seidl and Zorn combined profiling with a variety of different information sources present at the time of object allocation to predict an object's reference frequency and lifetime [64]. They showed that program references to heap objects are highly predictable. These studies focused on a program's paging behavior, not its cache behavior. Our work differs, not only because of the vast difference in cost between a cache miss and a page fault, but also because cache blocks are far smaller than memory pages.

Calder et al. [12] applied placement techniques developed for instruction caches [29, 58, 51] to data. They used a compiler-directed approach that creates an address placement for the stack (local variables), global variables, heap objects, and constants in order to reduce data cache misses. Their technique, which requires a training run to gather profile data, shows little improvement for heap objects but significant gains for stack objects and globals. By contrast, we provide techniques and tools for cache-conscious layout of heap-allocated structures that produce significant improvement without profiling. In addition, they used an entirely different allocation strategy, based on a history of the previously allocated object, rather than the programmer-supplied hint that `ccmalloc` uses to co-locate objects.

Above all things, keep clean.
It is not necessary to be a
pig in order to raise one.
—*R. G. Ingersoll,* About Farming in Illinois

Chapter 5

Using Garbage Collection to Optimize Data Layout

Chapter 4 showed that dynamically reorganizing data structures while a program executes can increase its reference locality and improve its cache performance. Garbage collectors, which support automatic memory management, determine when dynamically allocated storage has become unreachable and automatically recycle that memory. One type of garbage collector—a copying garbage collector—does this 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. 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.

This chapter discusses how a copying garbage collector can produce or improve a cache-conscious structure layout using program profiles. 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 our approach, a program is instrumented to profile its 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. The garbage collector uses a profile to construct an object affinity graph, in which weighted edges encode the temporal affinity between objects (nodes). A new copying garbage collection algorithm uses the affinity graph to produce cache-conscious data layouts while copying objects. 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%.

Earlier research attempted to improve a program's virtual memory (page) locality by changing the traversal algorithm used by a copying garbage collector [52, 82, 42, 23]. We compare our cache-conscious copying scheme against one such algorithm (the Wilson-Lam-Moher algorithm [82]). 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.

This chapter has six sections. Section 5.1 describes generational copying garbage collection and provides an overview of the most common traversal algorithm for copying objects. Section 5.2 discusses the design and implementation of our low overhead real-time data profiling system for object-oriented programs. Section 5.3 describes how this profiling information is used to construct object affinity graphs. Section 5.4 explains our

```
scavenge()
{
   Flip roles of Fromspace, Tospace;
   unprocessed = free = Tospace;
   for R in root set
      R = Tospace_copy(R);
   while unprocessed < free
   {
      for P in children(unprocessed)
         *P = Tospace_copy(*P);
      unprocessed +=
         sizeof(*unprocessed);
   }
}
```

```
Tospace_copy(P)
{
   if forwarded(P)
      return forwarding_addr(P);
   else
   {
      addr = free;
      copy(P, free);
      free += sizeof(P);
      forwarding_addr(P) = addr;
      return addr;
   }
}
```

Figure 5-1.   Cheney's copying algorithm.

cache-conscious copying algorithm. Section 5.5 presents experimental results that illustrate the benefits of this approach. Section 5.6 surveys related work.

5.1 Background: Generational Copying Garbage Collection

This section provides a brief overview of generational copying garbage collection. It first discusses copying garbage collection and then describes generational copying garbage collection. Jones and Lins [37] is a good reference for more information on generational copying and other types of garbage collectors.

Copying collectors divide the heap equally into two semi-spaces, one of which contains current data (*TO space*), and the other garbage (*FROM space*). Garbage collection starts by flipping the roles of the two spaces. Starting from the roots, which are data objects that are known to be live (for e.g., global variables, stack data, variables in registers), the copying collector traverses the live data structures in *FROM space* and copies them to *TO space*.

A common traversal algorithm for copying objects into *TO space* is Cheney's algorithm [18] (see Figure 5-1). Starting with the root set, objects are traversed in breadth-first order and copied to *TO space* as they are visited. Breadth-first traversal requires a queue.

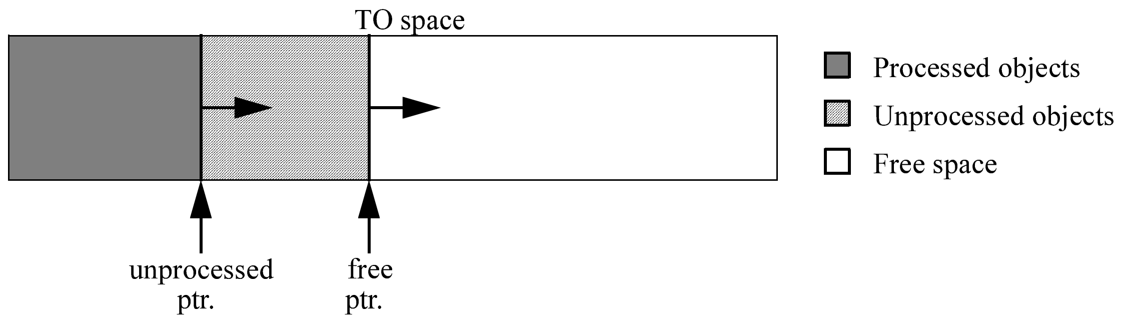


Figure 5-2. TO space during scavenging.

Objects to be processed are extracted from the head of the queue, while their children (if any) are added to the tail of the queue. The algorithm terminates when the queue is empty.

Cheney's algorithm does not use extra space to maintain the queue. Rather, it uses an elegant technique illustrated in Figure 5-2 which utilizes two pointers (*unprocessed* and *free*). Since the algorithm copies objects as they are visited, it uses these *TO space* copies as queue elements for breadth-first traversal. The head and tail of the queue are marked by the *unprocessed* and *free* pointer, respectively. Once an object is processed, it is removed from the head of the queue by incrementing the *unprocessed* pointer, and any children it may have are added to the tail of the queue by copying them to *TO space* and incrementing the *free* pointer.

Garbage collection ends after all live data in *FROM space* has been traced and copied to *TO space*. Since garbage is simply abandoned in *FROM space*, copying collectors are often described as scavengers [78]—they pick out and take away the useful data amidst the garbage.

Copying garbage collectors have some disadvantages. The most important drawback is that since all live data must be traversed and copied, garbage collection may cause obtrusive program pauses. Generational garbage collectors attempt to address this shortcoming. The generational hypothesis is that most objects die young [78], and garbage collection can be made much more efficient by concentrating effort on reclaiming data most likely to be garbage (i.e., young objects). Much research has gathered evidence that supports this hypothesis.

The generational strategy is to segregate data by age into multiple heap regions called generations. Different generations can then be collected at different frequencies, with the youngest generation collected most often, and older generations collected less frequently. Figure 5-3 illustrates the heap organization from the garbage collector's viewpoint for one kind of generational copying garbage collector [36]. The youngest (first) generation holds the most recently allocated objects. Objects that survive repeated scavenges are promoted to older (higher) generations. Garbage collection activity focuses on young objects, which typically die faster than old objects. Each generation is divided into one or more steps that encode objects' age. The first step of a generation is the youngest. Objects that survive scavenges are moved to the next step. Objects in the oldest step of a generation are promoted to the youngest step of the next generation. Each step consists of a collection of fixed size blocks, which are not necessarily contiguous in memory.

The scavenger always collects a generation *g* and all generations younger than *g*. Collecting a generation involves copying all objects in the generation that are reachable from the roots (objects in the generation pointed to by objects in older generations) into free blocks. The blocks that previously held the generation can be reused. The new space to which generation objects are copied is called *TO space* and the old space is called *FROM space* [27].

Ungar and Jackson [79] demonstrated performance advantages from not copying large objects. The UMass garbage collector toolkit [36], which we use for our experimental evaluation, provides a separate *large object space* (LOS) as part of the collected area. Each step has an associated set of large objects (≥ 256 bytes) that are of the same age as the small objects in the step. A step's large objects, though logically members of the step, are never physically moved. Instead, they are threaded onto a doubly linked list and moved from one list to another. When a large object survives a collection, it is unlinked

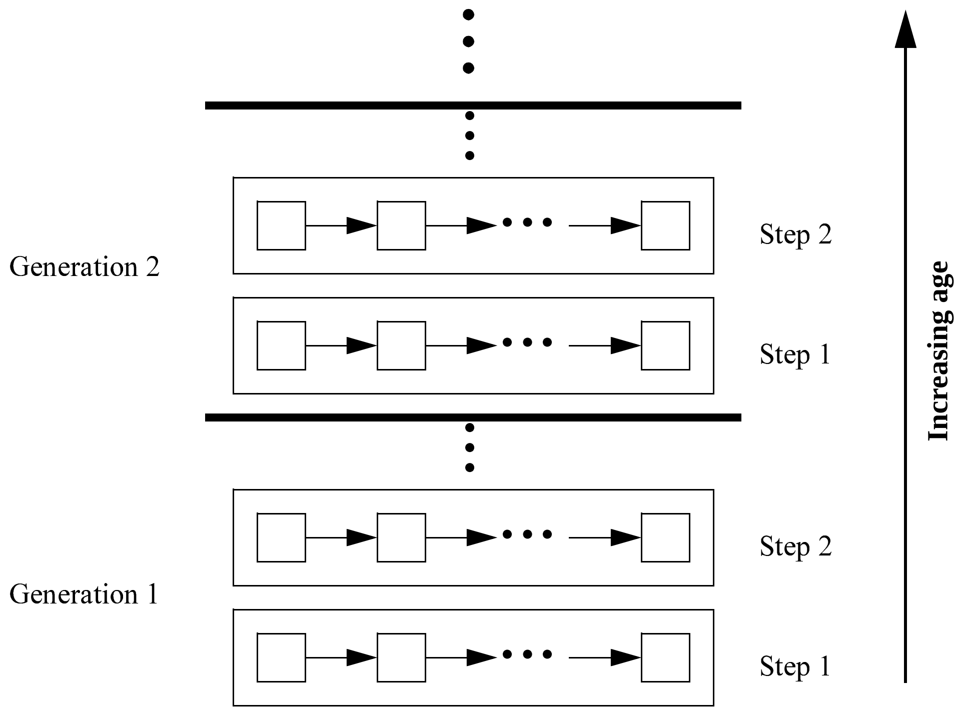


Figure 5-3. Heap layout for a generational garbage collector.

from its current step's list and added to the *TO space* list of the step to which it is promoted. The toolkit does not compact the large object space.

5.2 Low Overhead Real-Time Data Profiling

Cache-conscious data placement requires knowledge of data-access patterns to be effective. In particular, it requires information about contemporaneously accessed data items. These can be placed next to each other, so that they are likely to reside in the same cache block. In addition, since cache blocks are fairly small, the information concerning contemporaneously accessed data items must be quite accurate for it to be useful. This requires identifying specific structure instances that should be co-located. In general, static program analyses cannot provide this level of detail. Hence, we profile a program's data accesses.

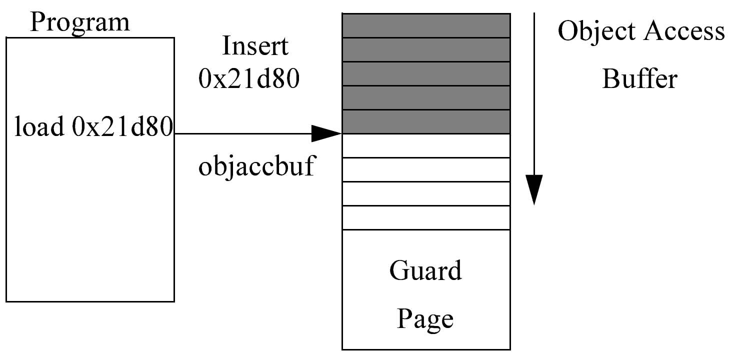


Figure 5-4. Object access buffer.

A profile of an earlier training run is commonly used to guide program optimizations. However, data access patterns require real-time profiling (i.e., profiling the current program execution) because of the difficulty of providing object names that are consistent and usable between runs of a program. Real-time profiling also spares a programmer an extra profile-execute cycle, as well as the difficulty of finding representative training inputs. However, the overhead of real-time profiling must be low enough that the performance improvements are not outweighed by profiling costs. The rest of this section discusses the design and implementation of a low-overhead, real-time data-access profiler that provides sufficient information for cache-conscious data placement.

In the most general case, profile-guided data placement requires tracing every load and store to heap data. The overhead of such tracing (a factor of 10 or more [12]) precludes its use in real-time profiling. However, two properties of object-oriented programs permit low overhead data profiling:

- most objects are small, often less than 32 bytes, and
- most object accesses are not lightweight.

Section 5.5 provides experimental results to support these assertions.

If most objects are small (e.g., less than 32 bytes), then it is not necessary for data profiling to distinguish different fields within the same object, since the objects reside entirely

```
ld baseobjptr, %reg
st %reg, [%objaccbuf]
add %objaccbuf, 4, %objaccbuf
```

Figure 5-5. Profiling instrumentation code for the load of a base object address.

within a cache block. Profiling can be implemented at object, not field, granularity. This greatly reduces the number of program instructions that have to be instrumented. Moreover, if most object accesses are not lightweight (i.e., multiple fields are accessed together or an access involves a method invocation), then profiling instrumentation (several instructions per object access) will not incur a large overhead.

To perform profiling at object granularity, our real-time data profiling system only instruments loads of base object addresses. The base object address is the object's start address, and object fields are accessed using offsets from this address. The profiling system uses information provided by a slightly modified compiler, which retains object-type information until the code-generation phase to permit this selective instrumentation of loads. This optimization coalesces multiple, consecutive references to the same object into a single profiled event, and substantially reduces the profiling overhead. Since cache-conscious co-location is performed at object, not field, granularity, this loss of information is of no consequence.

Cache-conscious co-location requires information about contemporaneous object accesses. We use an object access buffer, which is a sequential structure, similar to the sequential store buffer used in the garbage collection toolkit (Figure 5-4), to collect this information. The instrumentation enters the base object address in this object access buffer. The sequential order of base object addresses in the buffer reflects the temporal ordering of a program's object accesses. Figure 5-5 shows the instrumentation emitted for the load of a base object address (assuming the object-access-buffer pointer is stored in a dedicated register).

```
construct_obj_affinity_graphs()
{
   limit = objaccbuf
   objaccbuf = OBJ_ACC_BUF_BASE;
   while(objaccbuf < limit)
   {
      insert_locality_queue(objaccbuf);
      if(!exists_obj_affinity_node(objaccbuf))
         create_obj_affinity_node(objaccbuf);
      increment_obj_affinity_edge_weights(objaccbuf);
      objaccbuf += 4;
   }
}
```

Figure 5-6. Constructing object affinity graphs.

The object access buffer is normally processed just before a scavenge to guide cache-conscious object co-location. However, it may overflow between scavenges. Rather than include an explicit overflow check in the instrumentation, the virtual-memory system causes a page trap on buffer overflow. This is a cost-effective solution since our experience indicates that setting the buffer size to 51,200 entries (200 KB) prevents overflow. The trap handler processes the buffer and restarts the application.

Finally, for reasons put forth in Section 5.4.2, our generational garbage collection scheme focuses on longer lived objects. Since most objects die young, profiling overhead can be reduced by instrumenting only those loads that correspond to older objects. However, it is not possible to precisely identify all older object loads statically. Instead, we apply a heuristic approach that works well in practice. Our static filtering technique uses dataflow analysis to identify loads that target newly allocated objects. These loads are not instrumented. This optimization reduced the number of instrumented loads by a factor of 3-10.

5.3 Constructing Object Affinity Graphs

As described in Section 5.1, generational garbage collection copies live objects to *TO space*. Our goal is to use data-profiling information to produce a cache-conscious layout of objects in *TO space* that places objects with high temporal affinity next to each other, so

```
insert_locality_queue(objaccbuf)
{
   if (in_locality_queue(objaccbuf))
   {
      move_to_queue_tail(objaccbuf);
   }
   else
   {
      if (is_queue_full())
         delete_queue_hd();
      insert_queue_tail(objaccbuf);
   }
}
```

Figure 5-7. Locality queue insertion.

that they are likely to be in the same cache block. While the data-profiling scheme discussed in the previous section captures the temporal ordering of base object addresses, this information is not in a form suitable for efficient processing. To address this, we use the data-profiling information to construct object affinity graphs. An object affinity graph is a weighted undirected graph in which nodes represent objects and edges are labelled with the number of times that objects are contemporaneously accessed.

Since generational garbage collection processes objects in the same generation together, we construct a separate affinity graph for each generation (except the first, see Section 5.4.2). This is possible because an object's generation is encoded in its address. Although this scheme precludes placing objects in two different generations in the same cache block, we choose this approach for two reasons. First, the importance of inter-generation object co-location is unclear. Second, the only way to achieve inter-generation co-location is to demote the older object or promote the younger object. Both alternatives have disadvantages. Since generational garbage collection copies all objects of a generation together, intra-generation pointers are not explicitly tracked. The only safe way to demote an object is to subsequently collect the generation it originally belonged to, in order to update any pointers to the demoted object, which can produce unacceptably long garbage collection times. The other option is to promote the younger object. Such promotion is safe since the younger object's generation is being collected (this will update any

```
increment_obj_affinity_edge_weights(objaccbuf)
{
   queue_elem = NULL;
   init_locality_queue();
   do
   {
      queue_elem = next_queue_elem();
      if(exists_obj_affinity_edge(queue_elem, queue_tail())
         increment_affinity_edge_weight(queue_elem, queue_tail());
      else
         add_affinity_edge(queue_elem, queue_tail());
   } while (queue_elem != queue_tail())
}
```

Figure 5-8. Incrementing affinity graph edge weights.

intra-generation pointers to the object) Moreover, generational collectors track pointers from older objects to younger objects, so they can be updated (at a possibly high processing cost). However, the locality benefit of this promotion will not start until the older generation is collected (since it cannot be co-located with the older object until that time), which may be much later. In addition, there is the danger of premature promotion if the younger object does not survive long enough to merit promotion.

Figure 5-6, Figure 5-7, and Figure 5-8 contain the algorithm used to construct object affinity graphs (one per generation) from profile information. The size of the locality queue used in the algorithm is an important parameter. Too small of a queue runs the risk of missing important temporal relationships, but a large queue can result in huge object affinity graphs and long processing times. We used a queue size of three, since experiments (see Table 5.5) indicated that it gives the best results. Given the sizes of objects (32 bytes or less) relative to cache blocks (64 bytes), it is rarely possible to pack more than three objects in a cache block. Hence, it is not possible to take advantage of the richer temporal relationship information that bigger queue sizes offer.

Prior to each scavenge, the object affinity graphs can either be re-created anew from the contents of the object access buffer, or the profiling information can be used to update existing graphs. The suitability of these approaches depends on application characteristics. Applications with phases that access objects in distinct manners could benefit more from re-creation (provided phase durations are longer than the interval between scavenges), whereas applications with uniform behavior might be better suited to the incremental approach. Our implementation re-creates the object affinity graph prior to initiating a scavenge. This permits demand-driven graph construction that builds graphs only for the generations that are going to be collected during the subsequent scavenge. In addition, this re-creation approach minimizes the amount of garbage that is incorrectly copied (see Section 5.4.2).

5.4 Combining Cache-Conscious Data Placement with Garbage Collection

Cheney's algorithm copies objects to *TO space* in breadth-first order. Moon describes a modification to this algorithm that results in approximate depth-first copying [52]. Wilson et al. further refine the traversal to obtain hierarchical grouping of objects in *TO space* [82]. The copying algorithm (Figure 5-9) described in this section determines the traversal it performs dynamically using the object affinity graph.

5.4.1 Cache-Conscious Copying Algorithm

Our cache-conscious copying algorithm can be divided into three steps.

STEP 1: Flip the roles of *FROM space* and *TO space*. Initialize the *unprocessed* and *free* pointers to the beginning of *TO space*. From the set of roots present in the affinity graph, pick the one with the highest affinity edge weight. Perform a greedy depth-first traversal of the entire object affinity graph starting from this node (i.e., visit the next unvisited node connected by the edge with greatest affinity weight). The stack depth for the depth-first traversal is limited to the number of nodes in the object affinity graph, and hence the object access buffer can be used as a scratch area for this purpose. In parallel with this greedy depth-first traversal, copy each object visited to *TO space* (increment the *free* pointer). Store this new object address as a forwarding address in the *FROM space* copy of the object. After this step all nodes of the object affinity graph will be laid out in

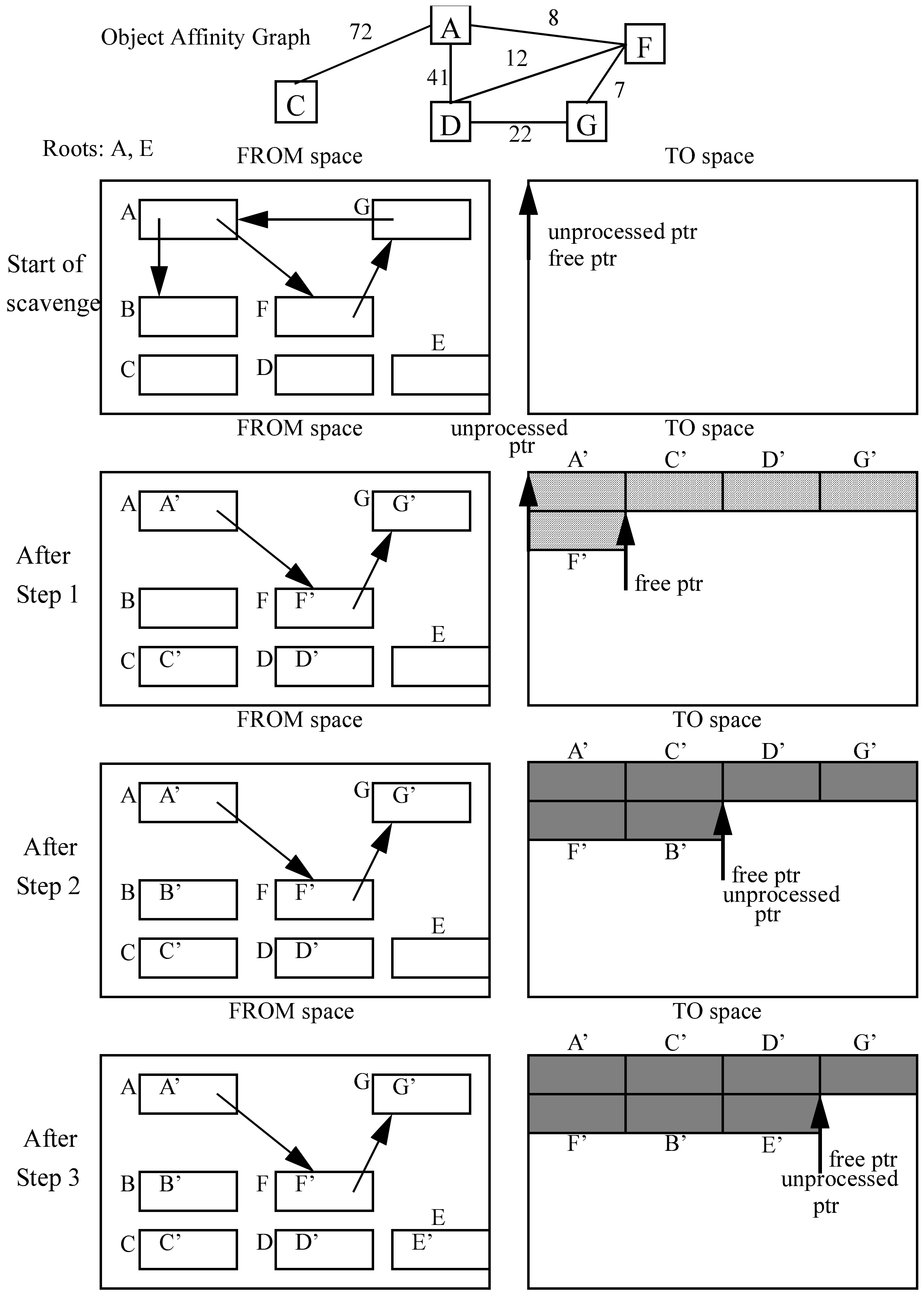


Figure 5-9. Combining cache-conscious data placement with garbage collection.

TO space in a manner reflecting object affinities (Figure 5-9), but will still contain pointers to objects in *FROM space*.

STEP 2: All objects between the *unprocessed* and *free* pointers are processed using Cheney's algorithm (except the step of copying the roots). We did not use a depth-first copying algorithm for this step and the next, since improving locality among infrequently accessed objects (those that do not appear in the affinity graph) is unlikely to compensate for the higher overhead incurred by the depth-first traversal.

STEP 3: This is a cleanup step where the root set is examined to ensure that all roots are in *TO space* (this is required as all roots may not be present in the object affinity graph or reachable from these objects). Any roots not present are copied to *TO space* and processed using Cheney's algorithm (Figure 5-9).

5.4.2 Discussion

The first step of the algorithm copies objects by traversing the object affinity graph, which may retain objects not reachable from the roots of the generation (i.e., garbage). However, since the system recreates the object affinity graph from new profile information prior to each scavenge, such garbage will be incorrectly promoted at most once. In addition, we focus our cache-conscious data placement efforts on longer-lived objects and do not use our copying algorithm in the youngest generation (where new objects are allocated and most of the garbage is generated). Table 5.6 demonstrates that the amount of garbage copied is negligible.

The copying algorithm described performs a greedy depth-first traversal of the object affinity graph. This performed better than several alternative traversal methods that we explored, such as greedy breadth-first traversal and depth-first traversal with lookahead. The depth-first-with-lookahead traversal attempts to maximize the combined affinity weights of the next two objects visited. The greedy-depth first traversal scheme appears to outperform other schemes due to its ability to capture important temporal relationships (object affinity graph edge weights were highly skewed, making the greedy choice often the correct choice) at a low processing cost. In addition, the size of objects relative to cache blocks prevents more sophisticated algorithms from exploiting their potentially better layouts to compensate for larger processing costs. Table 5.7 contains the experimental data.

5.5 Experimental Evaluation

This section presents experiments performed to support our assumption that object-oriented programs manipulate small objects (less than 32 bytes), to demonstrate that our real-time data-profiling technique incurs low overhead, and finally, to measure the impact of our cache-conscious object layouts on program performance.

5.5.1 Experimental Methodology

Our system uses the Vortex compiler infrastructure developed at the University of Washington [17]. Vortex is a language-independent optimizing compiler for object-oriented languages, with front ends for Cecil, C++, Java, and Modula-3. In addition, both Cecil and Java support generational garbage collection. This system uses the University of Massachusetts language-independent garbage collector toolkit [36]. The toolkit implements a flexible generation scavenger [48, 78] with support for a time-varying number of generations of time-varying size (see Figure 5-3). The collector was modified to incorporate our cache-conscious copying scheme.

Java is a widely popular object-oriented language [30]. Cecil [15, 16] is a dynamically-typed, purely object-oriented language. It combines multi-methods with a simple classless object model, a kind of dynamic inheritance, and modules. Instance variables in Cecil are accessed solely through messages, and can be replaced or overridden by methods. The Cecil and Java benchmark programs used in the experiments are described in Table 5.1. The first five are Cecil programs and the rest are Java programs. The programs were compiled at the highest optimization level (o2), which applies techniques such as class analy-

Program	Lines of Code[a]	Description
richards	400	Operating system simulation
deltablue	650	Incremental constraint solver
instr sched	2,400	Global instruction scheduler
typechecker	20,000[b]	Typechecker for old Cecil type system
new-tc	23,500[b]	Typechecker for new Cecil type system
cassowary	3,400	Linear constraint solver
espresso	13,800	Martin Odersky's drop-in replacement for javac
javac	25,400	Sun's Java source to bytecode compiler
javadoc	28,500	Sun's documentation generator for Java source
pizza	27,500	Pizza to Java bytecode compiler

Table 5.1: Benchmark programs.

a. Plus an 11,000 line standard library for the Cecil programs and a 13,700 line standard library (JDK 1.0.2) for the Java programs.

b. The two Cecil typecheckers share approximately 15,000 lines of common support code, but the type-checking algorithms are completely separate and were written by different people.

sis, splitting, class hierarchy analysis, class prediction, closure delaying, and inlining, in addition to traditional optimizations [17].

The experiments were run on a single processor of a Sun Ultraserver E5000, which contained 12 167Mhz UltraSPARC processors and 2GB of memory, running Solaris 2.5.1. The large amount of system memory ensures that locality benefits are due to improved cache performance and not paging activity. This system has two levels of data cache—a 16 KB direct-mapped level 1 cache with 16 byte cache blocks, and a unified (instruction and data) 1 MB direct-mapped level 2 cache with 64 byte cache blocks. The system has a 64 entry iTLB, as well as a 64 entry dTLB, both of which are fully associative. A level 1 data cache hit takes one processor cycle. A level 1 cache miss, followed by a level 2 cache hit, costs 6 additional cycles. A level 2 cache miss typically results in an additional 64

Program	**# of heap allocated small objects (< 256 bytes)**	**Bytes allocated (small objects)**	**Avg. small object size (bytes)**	**# of heap allocated large objects (>= 256 bytes)**	**Bytes allocated (large objects)**	**% small objects**
richards	567,896	4,551,792	**8.0**	2	2,064	**100**
deltablue	4,575,532	40,173,296	**8.8**	2	2,064	**100**
instr sched	783,929	7,276,792	**9.3**	31	50,912	**100**
typechecker	14,095,598	118,520,372	**8.4**	1,821	1,676,104	**100**
new-tc	13,023,528	112,296,720	**8.6**	1,268	1,155,276	**100**
cassowary	958,355	19,016,272	**19.8**	6,094	2,720,904	**99.4**
espresso	287,209	8,461,896	**29.5**	1,583	1,761,104	**99.5**
javac	489,309	15,284,504	**31.2**	2,617	1,648,256	**99.5**
javadoc	359,746	12,598,624	**35.0**	1,605	1,158,160	**99.6**
pizza	269,329	7,739,384	**28.7**	1,605	1,696,936	**99.4**

Table 5.2: Most heap-allocated objects are small (less than 32 bytes).

cycle delay. Each experiment was repeated five times and the average value reported (in all cases the variation between the smallest and largest values was less than 2%).

The generational garbage collector was configured as follows. The first three generations were each limited to 512 KB and the fourth and final generation was initially set to 4MB but could grow if needed. This configuration permitted the youngest generation to fit completely in the 1 MB level 2 cache. Previous research has indicated that this is important for good garbage collection performance [83]. Garbage collections were triggered if any generation had insufficient space for objects. The first generation, in which new objects were allocated, used a Cheney-like copying scheme (to minimize copying garbage—see Section 5.4.2), while the other generations used the object affinity graph co-location scheme described in Section 3.1.

5.5.2 Experimental Results

Our first set of experiments were designed to verify our conjecture that most heap-allocated objects are small. The run-time system was instrumented to gather object allocation statistics. Table 5.2 shows the results of these experiments. Objects larger than 256 bytes are allocated in a separate large object space (see Section 5.1), and are never physically moved. Fortunately, the results indicate that the vast majority of objects are smaller than 256 bytes. In addition, the average size of these small objects ranges from 8–35 bytes.

However, small objects often die fast. Since our cache-conscious layout technique is only effective for longer-lived objects, which survive scavenges, we are more interested in live object statistics. Table 5.3 shows the results of the next experiment, which measured the number of small objects that were live after each scavenge (all objects belonging to any generation not collected during the scavenge were considered live), averaged over the entire program execution. Once again, the results support our hypothesis that most objects are small (i.e., not larger than 32 bytes on average).

The next set of experiments measured the instrumentation overhead of our real-time data profiling with no co-location applied (Table 5.4). The results indicate that the overhead of our real-time data-profiling technique is low (less than 6%). Next, we measured the impact of the queue size used to construct object affinity graphs from object access buffer information. This experiment has the garbage collector using the object affinity graph to perform object co-location. As Table 5.5 indicates, a queue size of three worked best for these set of benchmarks on our system. While bigger queue sizes help create object affinity graphs that capture more temporal relationships, these graphs take longer to process at garbage collection time. In addition, given the sizes of objects (32 bytes or less) relative to cache blocks (64 bytes), it is rarely possible to pack more than three objects in a cache

Program	**Avg. # of live small objects**	**Bytes occupied (live small objects)**	**Avg. live small object size (bytes)**	**Large objects**	**% live small objects**
richards	645	9,926	**15.4**	2	**99.7**
deltablue	16,567	305,637	**18.5**	2	**100.0**
instr sched	6,456	157,736	**24.4**	31	**99.5**
typechecker	51,627	1,114,865	**21.6**	1,821	**96.5**
new-tc	58,858	1,392,212	**23.7**	1,268	**97.9**
cassowary	25,648	586,304	**22.9**	1,699	**93.8**
espresso	72,316	2,263,763	**31.3**	563	**99.2**
javac	64,898	2,013,496	**31.0**	194	**99.7**
javadoc	62,170	1,894,308	**30.5**	219	**99.6**
pizza	51,121	1,657,847	**32.4**	287	**99.4**

Table 5.3: Most live objects are small (less than 32 bytes).

block. Hence it is not possible to take advantage of the richer temporal relationship information that bigger queue sizes offer. The rest of the experiments use a queue size of three.

Our garbage collection scheme uses the object affinity graph to copy objects and as noted in Section 5.4.2 may copy garbage. Table 5.6 demonstrates that the amount of garbage copied is negligible.

Our next experiment tested alternative traversals of the object affinity graph, and measured their impact on overall performance. Table 5.7 contains the results. For reasons set forth in Section 5.4.2, the greedy-depth first traversal performs best.

We used the UltraSPARC's [72] hardware counters to measure the effect of our cache-conscious object layouts on cache miss rates. Table 5.8 contains measurements of the

Program	**Original execution time (secs)**	**Instrumented program execution time (secs)**	**% overhead of instrumentation**
richards	0.202	0.213	**5.45**
deltablue	3.369	3.544	**5.19**
instr sched	3.518	3.683	**4.69**
typechecker	347.352	358.467	**3.20**
new-tc	391.250	403.378	**3.10**
cassowary	34.46	36.15	**4.91**
espresso	44.94	47.20	**5.04**
javac	59.89	62.39	**4.17**
javadoc	44.42	46.25	**4.12**
pizza	28.59	29.97	**4.83**

Table 5.4: Overhead of real-time data profiling.

overall execution time (including the instrumentation and processing overhead of our technique). Our cache-conscious layouts reduce cache miss rates by 16–42% (our technique had practically no impact on L1 cache miss rates, as L1 cache blocks are only 16 bytes), producing corresponding reductions in execution times ranging from 10–37%, despite the technique's instrumentation and processing overhead.

Finally, we compared our approach against the Wilson-Lam-Moher algorithm [82], which uses a hierarchical decomposition algorithm for copying data between semi-spaces (instead of Cheney's breadth-first traversal) to improve a program's virtual memory (page) locality. This experiment (Table 5.9) investigated whether techniques designed to improve locality at the memory (page) level are effective at the cache level, and to ensure that the cache-miss rate reductions in Table 5.8 are not exaggerated by the poor locality of the base case (which uses Cheney's breadth-first traversal algorithm). Comparing

Program	**Queue size 2 (GC time)**	**Queue size 3 (GC time)**	**Queue size 5 (GC time)**	**Queue size 2 (Total time)**	**Queue size 3 (Total time)**	**Queue size 5 (Total time)**
richards	0.01	0.01	0.03	0.19	**0.17**	0.18
deltablue	0.57	0.73	1.17	2.97	**2.58**	2.87
instr sched	0.17	0.23	0.47	3.14	**2.76**	2.96
typechecker	26.49	37.63	63.97	317.22	**238.18**	259.42
new-tc	28.65	40.11	72.26	359.73	**247.62**	271.13
cassowary	2.67	3.13	5.02	32.51	**27.67**	28.96
espresso	8.18	9.35	14.17	42.16	**40.67**	44.29
javac	8.76	10.64	15.84	57.99	**53.18**	58.03
javadoc	7.02	8.40	12.16	43.23	**39.26**	42.22
pizza	3.90	4.87	8.33	27.92	**25.78**	28.45

Table 5.5: Impact of queue size on execution time.

Program	**Avg. # of live small objects (base)**	**Avg. # of live small objects (CCDP)**	**% garbage copied**
richards	645	647	0.31%
deltablue	16,567	16,578	0.07%
instr sched	6,456	6,471	0.23%
typechecker	51,627	51,648	0.04%
new-tc	58,858	58,873	0.03%
cassowary	25,648	25,671	0.09%
espresso	72,316	72,391	0.10%
javac	64,898	64,937	0.06%
javadoc	62,170	62,195	0.04%
pizza	51,121	51,152	0.06%

Table 5.6: Amount of garbage incorrectly copied.

Program	Greedy breadth -first (GC time)	Greedy depth-first (GC time)	Depth-first with look-ahead (GC time)	Greedy breadth -first (Total time)	Greedy depth-first (Total time)	Depth-first with lookahead (Total time)
richards	0.01	0.01	0.02	0.19	**0.17**	0.18
deltablue	0.68	0.73	1.06	3.15	**2.58**	2.79
instr sched	0.24	0.23	0.43	3.31	**2.76**	2.91
typechecker	36.24	37.63	54.01	321.20	**238.18**	268.76
new-tc	40.32	40.11	63.72	364.36	**247.62**	275.04
cassowary	3.17	3.13	4.79	32.11	**27.67**	28.80
espresso	9.17	9.35	12.02	43.08	**40.67**	42.55
javac	10.44	10.64	14.16	58.41	**53.18**	56.05
javadoc	8.65	8.40	11.30	42.97	**39.26**	41.48
pizza	4.92	4.87	6.77	27.93	**25.78**	27.96

Table 5.7: Impact of the object affinity graph traversal algorithm on performance.

Table 5.8 and Table 5.9, we see that for three benchmarks (*richards*, *deltablue*, and *instr sched*), the Wilson-Lam-Moher algorithm performs worse than Cheney's algorithm, while slightly outperforming it for *typechecker*, *new-tc,* and all the Java programs. These surprising results are easily explained. Since the system has 2GB of memory, no application pages. In addition, the system has a 64 entry dTLB (which supports a 512KB working set), hence the only applications that might suffer dTLB misses are *typechecker*, *new-tc,* and the Java programs (see Table 5.3), which is consistent with our measurements. Since the Wilson-Lam-Moher algorithm is ineffective at reducing a program's cache miss rate, and has a slightly higher overhead than Cheney's algorithm, it performs worse for *richards*, *deltablue*, and *instr sched*.

Program	L2 cache miss rate (base)	L2 cache miss rate (CCDP)	% reduction (L2 miss rate)	Execution time (base)	Execution time (CCDP)	% reduction (execution time)
richards	1.3%	1.0%	**21.4**	0.202	0.173	**14.4**
deltablue	3.6%	2.4%	**32.6**	3.369	2.578	**23.5**
instr sched	5.4%	3.9%	**27.8**	3.518	2.756	**21.7**
typechecker	9.5%	5.9%	**37.6**	347.352	238.179	**31.4**
new-tc	9.8%	5.7%	**41.7**	391.250	247.622	**36.7**
cassowary	8.6%	6.1%	**29.1**	34.46	27.67	**19.7**
espresso	9.8%	8.2%	**16.3**	44.94	40.67	**9.5**
javac	9.6%	7.7%	**19.8**	59.89	53.18	**11.2**
javadoc	6.5%	5.3%	**18.5**	44.42	39.26	**11.6**
pizza	9.0%	7.5%	**16.7**	28.59	25.78	**9.8**

Table 5.8: Impact of cache-conscious object layout.

5.6 Related Work

White [81] first suggested using garbage collection to improve a program’s locality of reference. Researchers investigated two approaches to using a garbage collector to improve paging behavior of Smalltalk and LISP systems [52, 82, 42, 23]. Static regrouping uses the topology of heap-allocated data structures to rearrange structurally related objects [52, 82], while dynamic regrouping [23] clusters objects according to a program’s data access pattern. Moon found that depth-first copying generally yields better virtual-memory performance than breadth-first copying for LISP, because it is more likely to place parents and offspring on the same page, particularly if data structures tend to be shallow, but wide [52]. Wilson et al. treated hash tables, which group data in a pseudo-random order, specially, and ‘normal’ data structures were copied in depth-first order [82]. Their results showed a significant reduction in the incidence of page faults. However, in a

Program	L2 cache miss rate (WLM)	L2 cache miss rate (CCDP)	% reduction (L2 miss rate)	Execution time (WLM)	Execution time (CCDP)	% reduction (execution time)
richards	1.3%	1.0%	**20.2**	0.211	0.173	**18.0**
deltablue	3.4%	2.4%	**29.6**	3.437	2.578	**25.0**
instr sched	5.3%	3.9%	**26.3**	3.621	2.756	**23.9**
typechecker	9.3%	5.9%	**36.1**	321.433	238.179	**25.9**
new-tc	9.6%	5.7%	**40.7**	358.512	247.622	**30.9**
cassowary	8.5%	6.1%	**28.0**	33.95	27.67	**18.5**
espresso	9.6%	8.2%	**14.4**	44.25	40.67	**8.1**
javac	9.4%	7.7%	**18.4**	58.38	53.18	**8.9**
javadoc	6.4%	5.3%	**17.3**	43.24	39.26	**9.2**
pizza	8.9%	7.5%	**15.5**	28.30	25.78	**8.9**

Table 5.9: Comparison with the Wilson-Lam-Moher algorithm.

later study, the authors found that the optimal grouping of data structure elements was very dependent on the shape and type of the structure being copied [42]. While hierarchical decomposition performed well for trees, it was disappointing for other structures. Court's dynamic regrouping technique takes advantage of specialized hardware to support incremental garbage collection, which tends to move objects to *TO space* in program access order, and this can dramatically reduce the number of page faults [23]. 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. As our results indicate, techniques that improve a program's page locality, are not necessarily effective at the cache level. In addition, we attempt to lay out objects in program-access order using real-time data profiling information, rather than a single traversal order.

Recently, Calder et al. applied placement techniques developed for instruction caches to data [12]. They use a compiler-directed approach that creates an address placement for the stack (local variables), global variables, heap objects, and constants in order to reduce data cache misses. Their technique, which requires a training run to gather profile data, shows little improvement for heap objects, but significant improvement for stack objects and globals. By contrast, we use low overhead real-time data profiling and copying garbage collection to implement on-the-fly cache-conscious data placement, showing significant improvements for programs that manipulate heap-allocated data structures.

Ding and Kennedy explored two run-time transformations that improve the memory-hierarchy performance of irregular computations and report promising results [25]. However, since the application domain they studied consists of irregular Fortran applications, it is hard to compare their results with ours.

Definitions might be good things, if only
we did not employ words in making them.
—Rousseau

Chapter 6

Cache-Conscious Structure Definition

Chapters 4 and 5 showed that an effective way to mitigate the continually increasing processor-memory performance gap is to allocate or reorganize data structures in a manner that increases a program's reference locality and improves its cache performance. Cache-conscious data layout, which clusters temporally related objects into the same cache block or into non-conflicting blocks, can produce significant performance gains.

However, the techniques discussed in Chapters 4 and 5 (with the exception of the techniques for reducing cache conflicts) work best for structures smaller than half of a cache block, as they attempt to cluster multiple structure instances in the same cache block. To address this limitation and make previous techniques applicable to larger structures, this chapter continues the study of data placement optimizations along the orthogonal direction of reordering the internal layout of a structure or class's fields. The chapter explores two cache-conscious definition techniques—*structure splitting* and *field reordering*—that can improve the cache behavior of programs. In other words, previous techniques focused

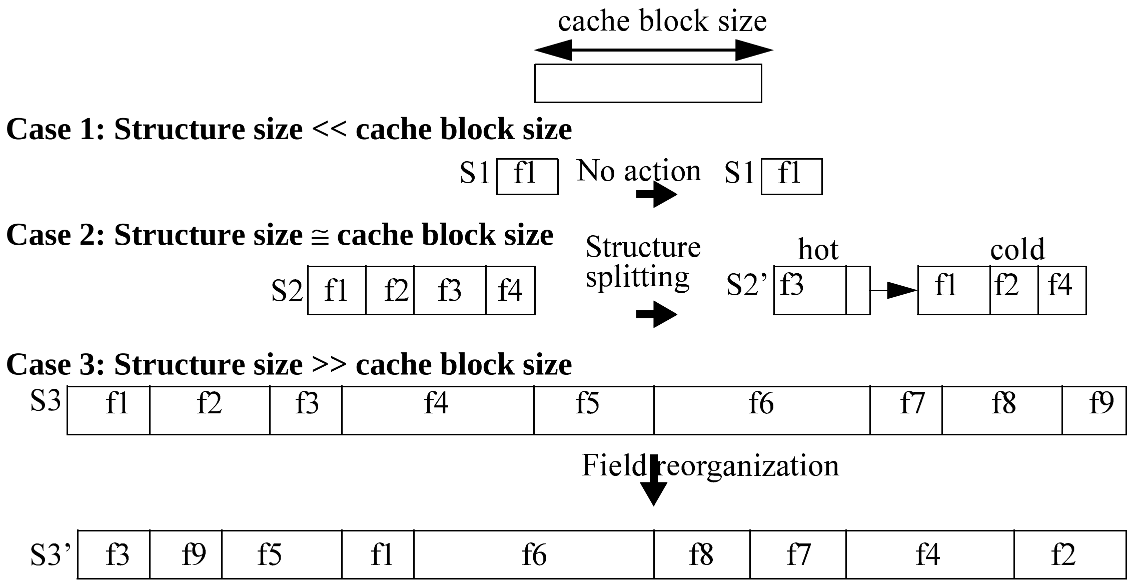


Figure 6-1. Cache-conscious structure definition.

on arranging structure instances in memory, while this chapter focuses on their internal organization.

Figure 6-1 illustrates the relationship of cache-conscious definition technique to the size of structure instances. For instances smaller than half a cache block (case 1) previous techniques are effective—such as the cache-conscious object co-location scheme described in the previous chapter, which uses a copying garbage collector to place objects referenced together near each other in memory. Hence, these structures are unlikely to benefit from additional manipulation at definition time. Such is the case for the Cecil objects of Chapter 5.

If the size of data structure elements is comparable to the size of a cache block (case 2), previous techniques that cluster multiple structure instances in the same cache block do not work well. However, reducing the effective structure instance size can permit application of these techniques. Chapter 2 discussed several complementary approaches to structure compression, such as data compression, pointer elimination, and structure splitting. Data compression does not appear well suited to this problem due to the compressiondecompression overhead each time a structure instance is accessed. Pointer elimination often requires programmer knowledge of the data structure, which makes it hard to automate. Structure splitting partitions structure elements into a hot and cold portion, based on field access frequencies. This can produce hot structure pieces smaller than a cache block, which permits application of cache-conscious reorganization techniques to these portions. In addition, for type-safe languages, structure splitting can be automated. Hence, Section 6.1 explores structure splitting as a mechanism to reduce effective structure size.

As Chapter 5 has shown, many Java objects satisfy this property (case 2). In addition, since Java is a type-safe language, class 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 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. To maximize the benefits of splitting, we carefully design our splitting algorithm to reduce these costs. In addition, we use our garbage collection scheme for cache-conscious object co-location (see Chapter 5) 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, 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.

Finally, when structure elements span multiple cache blocks (case 3), structure splitting is likely to produce hot instances that are larger than a cache block, making it ineffective.

In this case, 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. Yet, given the ever-increasing cache miss penalties, reordering structure fields to place those with high temporal affinity in the same cache block, is a relatively simple and effective way to improve program performance. Since this cannot be done automatically for many languages, our approach is to provide recommendations to the programmer on the order in which structure fields should occur.

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. To investigate the benefits of field reordering, this chapter describes an algorithm for recommending reorderings 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 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 [22].

The rest of the chapter is organized as follows. Section 6.1 investigates structure splitting. Section 6.2 discusses field reordering for C and describes our field reordering algorithm. Section 6.3 and Section 6.4 present our experimental results. Finally, Section 6.5 briefly discusses related work.

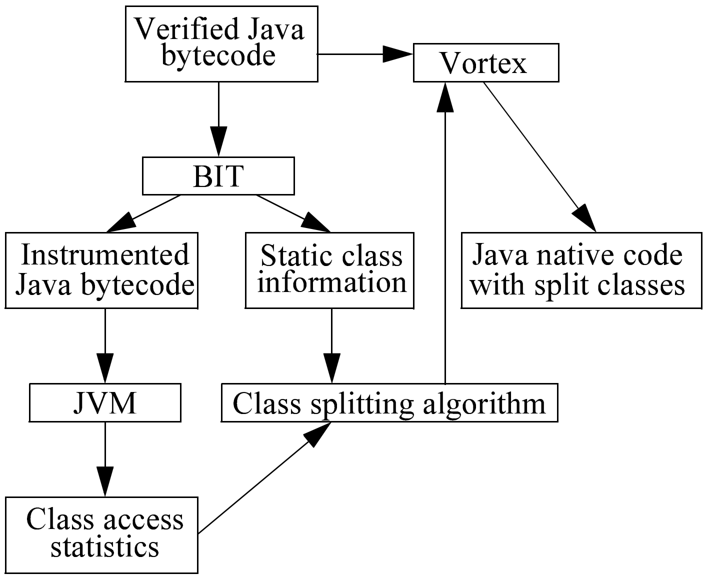

Verified Java bytecode
Vortex
BIT
Instrumented Java bytecode
Static class information
Java native code with split classes
JVM
Class splitting algorithm
Class access statistics

Figure 6-2. Class splitting overview.

6.1 Structure Splitting

Chapter 5 found that Java objects are small, but on average they are approximately 8 bytes larger than Cecil objects, primarily due to larger Java object headers. Directly applying the cache-conscious garbage collection co-location scheme to Java programs yields smaller performance improvements (10–20%) than those reported for Cecil (14–37%). A possible explanation for this difference is that larger Java objects reduce the number of contemporaneously accessed object instances that can be packed into a cache block.

One way to reduce the effective size of Java objects is to split Java classes into a hot (frequently accessed) portion and a cold (rarely accessed) portion, based on field access frequencies obtained via profiling. Splitting classes allows more hot object instances to be packed into a cache block and kept in the cache at the same time. Structure splitting is a well-known optimization that is often applied manually to improve performance. However, to the best of our knowledge, this is the first completely automatic implementation of the technique.

Figure 6-2 illustrates the class splitting process. First, a Java program, in the form of verified bytecodes, is statically analyzed and instrumented using BIT [47], which is a Java bytecode-instrumentation tool. Since standard library classes were not candidates for splitting to maintain program portability, the standard library was not instrumented. The static analysis produces a variety of class information, including class and field names, field types, and field sizes. Next, the instrumented Java program is executed and profiled. The profile measures class-instantiation counts and instance-variable-access statistics (for non-static class fields) on a per class basis. An algorithm uses the static and dynamic data to determine which classes should be split. Finally, these splitting decisions are communicated to the Vortex compiler [17], which compiles Java bytecode to native code. The compiler splits the specified classes and transforms the program to account for the change. The class splitting algorithm and associated program transformations are described in more detail in subsequent sections.

Applying the cache-conscious object co-location scheme described in the previous chapter to the Java programs obtained from class splitting results in performance improvements of 18–28%, with 22–66% of this improvement attributable to class splitting (see Section 6.3).

6.1.1 Class Information

BIT is used to gather static class information, including class name, number of non-static fields, and the names, access types (i.e., `private`, `protected`, `public`), and descriptors for all non-static fields. Non-static fields are tracked since these constitute the instance variables of a class and are allocated on the heap. In addition, BIT instruments the program to generate field-access frequencies on a per-class basis. While it may be possible to classify some member fields as hot or cold via static analysis, we profile a program to determine member-access frequency since this appears to be a simpler and more general approach. An instrumented program runs an order of magnitude slower than its original.

6.1.2 Hot/Cold Class Splitting Algorithm

Class splitting involves several trade-offs. Its primary advantage is the ability to pack more (hot) class instances in a cache block. Its disadvantages include the cost of an additional reference from the hot to cold portion, code bloat, more objects in memory, and an extra field and indirection for cold field accesses. This section describes a class splitting algorithm that considers these issues while selecting classes to split.

It is extremely hard to formulate the class splitting problem precisely, much less solve it optimally. The costs of splitting, such as code bloat and more objects in memory, are difficult to quantify as they are often non-deterministic and dependent on the specific system configuration. In addition, any precise solution will be valid only if the program is rerun on the same input data set and the program is deterministic. However, we are interested in splitting classes so the resulting program performs well for a wide range of inputs. For these reasons, the class splitting algorithm uses several heuristics. Measurements in Section 6.3.1 demonstrate that they work well in practice. In addition, they worked better than several alternatives that were examined. In the ensuing discussion, the term "field" refers to class instance variables (i.e., non-static class variables).

Figure 6-3 presents the splitting algorithm. The splitting algorithm only considers classes whose total field accesses exceed a specified threshold. This check avoids splitting classes in the absence of sufficient representative access data. The following formula worked well for determining this threshold. Let *LS* represent the total number of program field accesses, *C* the total number of classes with at least a single field access, F_i the number of fields in class *i*, and A_i the total number of accesses to fields in class *i*. The splitting algorithm only considers classes where:

$$A_i > LS / (100 * C) \qquad \text{EQ 1.}$$

These classes are called the 'live' classes. Increasing this threshold reduced the number of classes that were candidates for splitting in our benchmark suite. These included split classes that contributed to performance improvements. Reducing this threshold introduced a larger number of candidate classes. However, the additional split classes degraded performance when the programs were run on a different input data set than the one used to make the splitting decisions.

In addition, the splitting algorithm only considers classes that are larger than eight bytes and contain more than two fields. Splitting smaller classes is unlikely to produce any benefits, given the space penalty incurred by the reference from the hot portion to the cold portion.

Next, the algorithm labels fields in the selected ‘live’ classes as hot or cold. An aggressive approach that produces a smaller hot partition—and permits more cache-block co-location—also increases the cost of accessing cold fields. These competing effects must be balanced. Initially, the splitting algorithm takes an aggressive approach and marks any field not accessed more than $A_i / (2 * F_i)$ times as cold. Again, we experimented with different thresholds. Higher thresholds produced too many cold fields, which hurt performance. On the other hand, lower thresholds defeat the purpose of the algorithm, which is to be as aggressive as possible without degrading performance. If the cold portion of class *i* is sufficiently large to merit splitting (at least 8 bytes to offset the space required for the cold object reference), the following condition is used to counterbalance overaggressive splitting:

$$(max(hot(class_i)) - 2 * \Sigma\, cold(class_i)) / max(hot(class_i)) > 0.5 \qquad \text{EQ 2.}$$

where (in a slight abuse of notation) the *hot* and *cold* operators take as input a class, and return a sequence of access counts of a class’ hot and cold fields, respectively. This condition is motivated by trying to account for object co-location effects and can be informally justified as follows. Consider instances of two different classes, o_1 and o_2 (since typically two objects can be co-located in the same cache block), that are both comparable in size to a cache block and that have a high temporal affinity. Let instance o_1 have *n* fields that are accessed $a_1, .., a_n$ times, and o_2 have m fields that are accessed $b_1, .., b_m$ times. It is reasonable to expect the following access costs (number of cache misses) for the class instances o_1 and o_2, since in the worst case every field access incurs a cache miss, while in a more favorable case the accesses are clustered together, with the access count of the most frequently referenced field indicating the number of distinct times the instance is accessed:

$$max(a_1, ..., a_n) < cost(o_1) < \Sigma(a_1, ..., a_n)$$
$$max(b_1, ..., b_m) < cost(o_2) < \Sigma(b_1, ... b_m)$$

Now, if the hot portion of o_1 is co-located with the hot portion of o_2, and these fit in a cache block, then:

$$cost(o_1) + cost(o_2) \cong (max(hot(class_1), hot(class_2)) + \varepsilon) + 2 * (\Sigma cold(class_1) + \Sigma cold(class_2))$$

where ε is a very small quantity. This equation holds because the class instances have high affinity (hot field accesses are clustered) and the cold fields are accessed through a level of indirection. This will definitely be beneficial if the sum of the (best case) costs of accessing original versions of the instances is greater than the access cost after the instances have been split and hot portions co-located:

$$max(a_1, ..., a_n) + max(b_1, ..., b_m) > ((max(hot(class_1), hot(class_2)) + \varepsilon) + 2*(\Sigma cold(class_1) + \Sigma cold(class_2))$$

i.e.:

$$min(max(hot(class_1)), max(hot(class_2))) > 2 * (\Sigma cold(class_1) + \Sigma cold(class_2)) + \varepsilon$$

Since a priori we do not know which class instances will be co-located, the best we can do is to ensure that:

$$TD(class_i) = max(hot(class_i)) - 2 * \Sigma\ cold(class_i) >> 0 \qquad \text{EQ 3.}$$

This quantity is termed the '*temperature differential*' for the class. For classes that do not meet this criterion, a more conservative formula is used that labels fields that are

```
split_classes()
{
  for each class {
    mark_no_split;
    if((live)&&(suitable_size)){
      mark_fields_aggresive;
      if(sufficent_cold_fields)
        if(normalized_temperature_differential > 0.5)
          mark_split;
        else{
          re-mark_fields_conservative;
          if(sufficent_cold_fields)
            mark_split;
        }
    }
  }
}
```

Figure 6-3. Class splitting algorithm.

accessed less than $A_i / (5*F_i)$ as cold. If this does not produce a sufficiently large cold portion (greater than 8 bytes), the class is not split.

6.1.3 Program Transformation

We modified the Vortex compiler to split classes selected by the splitting algorithm and to perform the associated program transformations. Hot fields and their accesses remain unchanged. Cold fields are collected and placed in a new cold counterpart of the split class, which inherits from the primordial Object class and has no methods beyond a constructor. An additional field, which is a reference to the new cold class, is added to the original class, which now contains the hot fields. Cold fields are labelled with the `public` access modifier. This is needed to permit access to `private` and `protected` cold fields through the cold class reference field in the original (hot) class. Since these transformations are applied to verified bytecode they do not affect program security, provided that the compiler is a trusted entity.

Finally, the compiler modifies the code to account for split classes. These transformations include replacing accesses to cold fields with an extra level of indirection through the cold class reference field in the hot class. In addition, hot class constructors must first create a new cold class instance and assign it to the cold class reference field. Figure 6-4 illustrates these transformations for a simple example.

6.1.4 Discussion

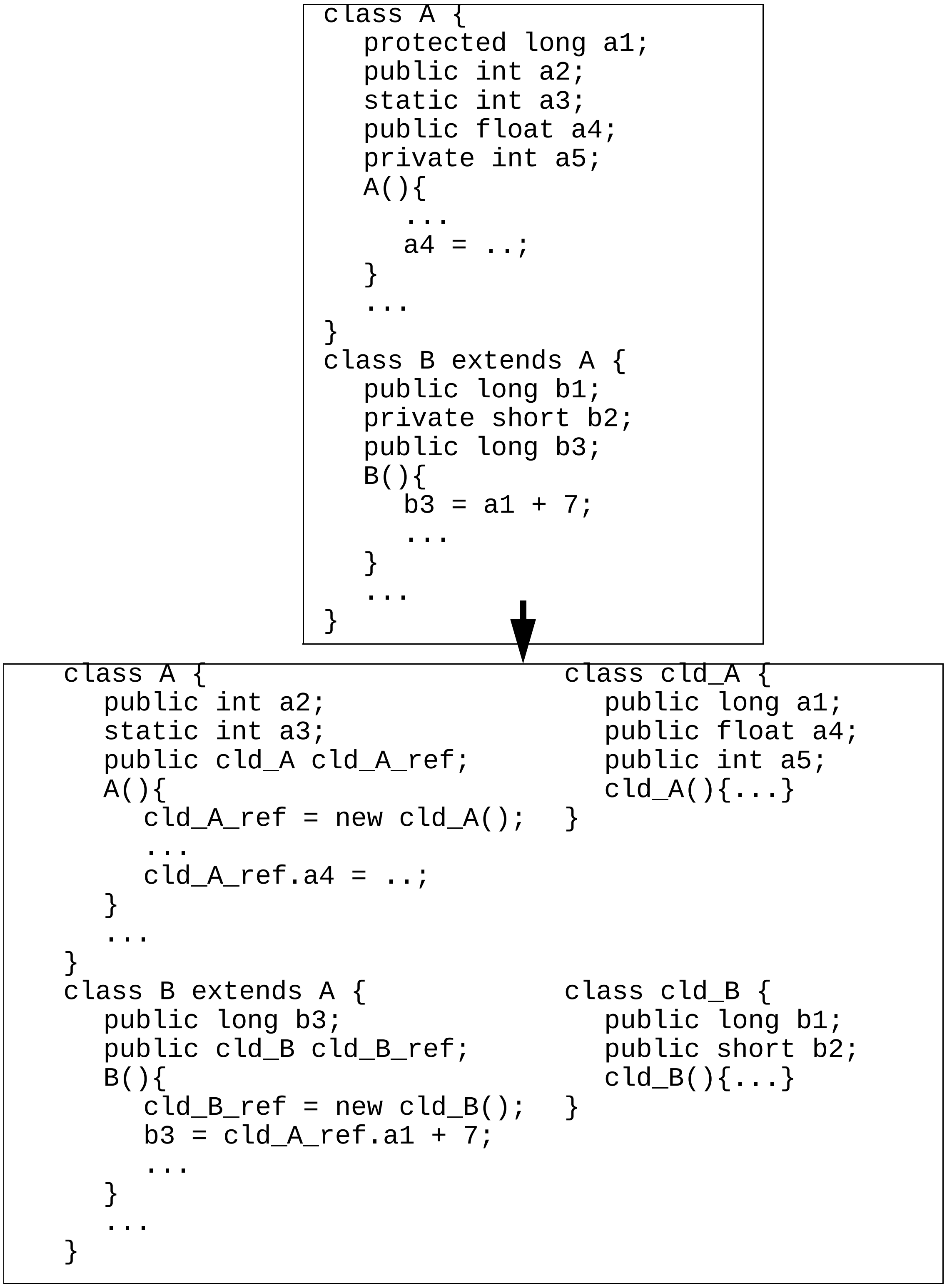
Some programs transfer structures back and forth to persistent storage or external devices. These structures cannot be transparently changed without losing backward compatibility. However, when new optimizations offer significant performance advantages, the cost of such compatibility may become high, and explicit input and output conversion necessary. Translation, of course, is the norm in languages, such as Java, in which structure layout is left to the compiler.

The splitting technique described produces a single split version of each selected class. A more aggressive approach would create multiple variants of a class, and have each direct subclass inherit from the version that is split according to the access statistics of the inherited fields in that subclass. To simplify our initial implementation, we choose not to explore this option, especially since its benefits are unclear. However, future work will investigate more aggressive class splitting.

Since this thesis focuses on improving data cache performance, class splitting only considers member fields and not methods. Method splitting could improve instruction cache performance. In addition, it offers additional opportunities for overlapping execution of mobile code with transfer [40].

6.2 Field Reordering

Commercial applications often manipulate large structures with many fields. Typically, fields in these structures are grouped logically, which may not correspond to their temporal access pattern. The resulting structure layout may interact poorly with a program's data access pattern and cause unnecessary cache misses. This section describes an algorithm



```
class A {
   protected long a1;
   public int a2;
   static int a3;
   public float a4;
   private int a5;
   A(){
      ...
      a4 = ..;
   }
   ...
}
class B extends A {
   public long b1;
   private short b2;
   public long b3;
   B(){
      b3 = a1 + 7;
      ...
   }
   ...
}
```

```
class A {                            class cld_A {
   public int a2;                       public long a1;
   static int a3;                       public float a4;
   public cld_A cld_A_ref;              public int a5;
   A(){                                 cld_A(){...}
      cld_A_ref = new cld_A();       }
      ...
      cld_A_ref.a4 = ..;
   }
   ...
}
class B extends A {                  class cld_B {
   public long b3;                      public long b1;
   public cld_B cld_B_ref;              public short b2;
   B(){                                 cld_B(){...}
      cld_B_ref = new cld_B();       }
      b3 = cld_A_ref.a1 + 7;
      ...
   }
   ...
}
```

Figure 6-4.   Program transformation.

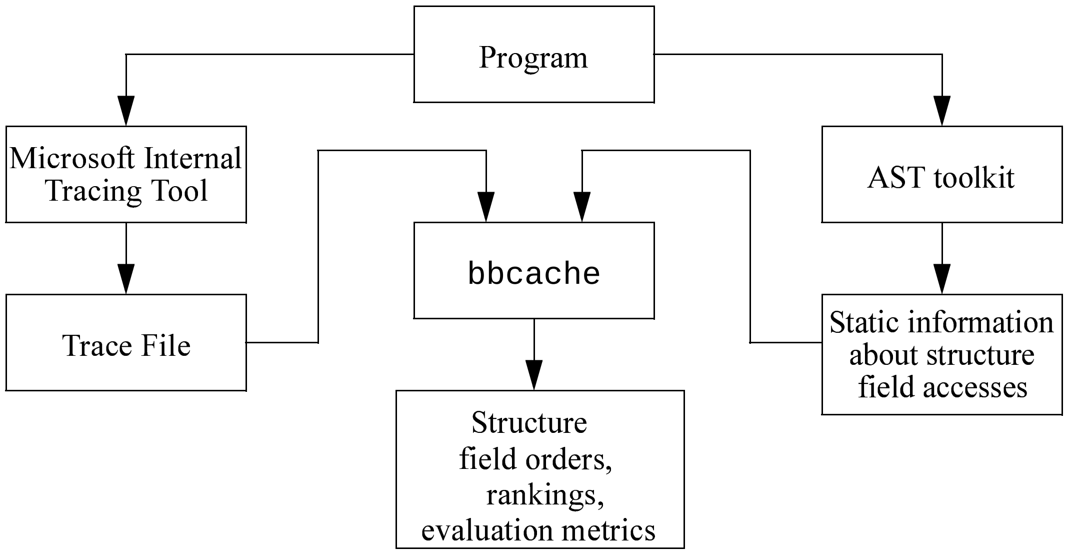

Program
Microsoft Internal Tracing Tool
AST toolkit
bbcache
Trace File
Static information about structure field accesses
Structure field orders, rankings, evaluation metrics

Figure 6-5. Field reordering overview.

for producing recommendations for reordering structure fields and incorporates this algorithm into a tool—`bbcache`. The field reordering recommendations attempt to increase cache block utilization, and reduce cache pressure, by grouping fields with high temporal affinity in a cache block.

For languages, such as C, that permit almost unrestricted use of pointers, reordering structure fields can affect program correctness—though this is often a consequence of poor programming practice. Moreover, C structures can be constrained by external factors, such as file or protocol formats. For these reasons, the field reordering recommendations must be examined by a programmer before they can be applied to C programs.

6.2.1 Field Reordering Overview

Figure 6-5 illustrates the field reordering process. A program is first profiled to create a record of its memory accesses. The trace file contains temporal information and execution frequency for structure field accesses. `bbcache` combines this dynamic data with static analysis of the program source to produce recommendations for reordering structure fields.

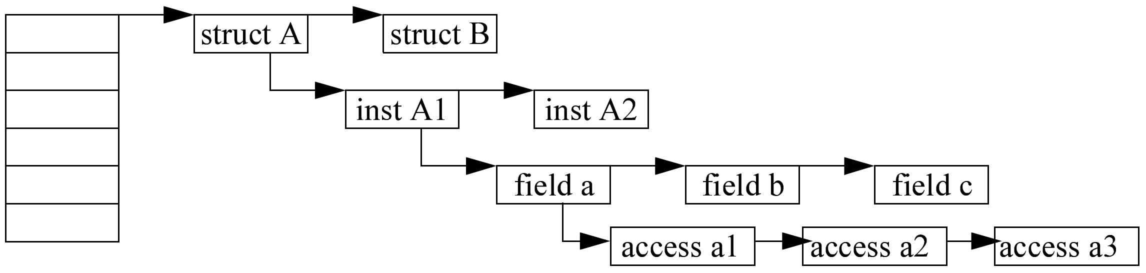


Figure 6-6. Structure access database.

The algorithm used to recommend structure field orders can be divided into three steps. First, construct a database containing both static (source file, line, etc.) and dynamic (access count, etc.) information about structure field accesses. The static and dynamic information complement each other, and provide more data for making field reordering decisions. Next, process this database to construct an instance field affinity graph for each structure instance. An instance field affinity graph is a weighted undirected graph in which nodes represent structure fields and edges encode temporal affinity between fields. Then combine these instance field affinity graphs to produce a structure field affinity graph. Finally, produce the recommended field order from these structure field affinity graphs.

`bbcache` also contains an evaluation facility that produces a cost metric, which represents a structure's cache block working set, and a locality metric, which represents a structure's cache block utilization. These metrics help compare the recommended field order against the original layout. They, together with a ranking of active structures based on their temporal activity and access frequency, can be used to identify structures most likely to benefit from field reordering. This is especially important for large commercial applications that have thousands of structures.

6.2.2 Constructing the Structure Access Database

The ASTtoolkit [24], a tool for querying and manipulating a program's abstract syntax tree, is used to analyze the source program. It produces a file containing information about each structure field access, including the source file and line at which the access occurs;

whether the access is ‘read’, ‘write’, or ‘read-write’; the field name; the structure instance; and the structure (type) name. A structure instance is a function-name/structure-name pair, where the function name corresponds to the function in which the instance is allocated. With pointer aliasing, computing structure instances statically in this manner is an approximation. The following example helps illustrate the problem. Consider consecutive accesses to fields *a* and *b* in two different structure instances (though indistinguishable with our approximation). This could lead to incorrectly placing fields *a* and *b* next to each other. However, this did not appear to be a serious problem for our purposes, since most instances showed similar access characteristics (i.e., consecutive accesses to the same field in different (indistinguishable) instances, rather than different fields). bbcache reads this file and builds a structure access database, which it represents as a hash table on structure names (Figure 6-6). Each hash table entry represents a structure type and points to a list of structure instances. Every structure instance points to a list of fields that were accessed through that instance, and each field in turn points to a list of access sites that record the source location from which the access took place. In addition, the program is profiled to collect temporal information (when in the program were the fields accessed) and execution frequency (how often were they accessed) of structure field accesses. bbcache uses program debugging information to associate temporal information and execution frequencies, from the program trace, with each field-access site.

6.2.3 Processing the Structure Database

The structure database contains information about field accesses for many instances of the same structure type. For each structure instance, bbcache constructs a field affinity graph, which is a weighted graph whose nodes represent fields and whose edges connect fields that are accessed together according to the temporal trace information. Fields accessed within 100 milliseconds of each other in the trace were considered to be accessed contemporaneously. While we experimented with several intervals ranging from 50-1000 ms, most structures did not appear to be very sensitive to the exact interval used to define contemporaneous access, and the results reported in Section 6.4 correspond to a 100ms

```
for each structure type
{
  for each instance of this type
  {
    combine field access information for multiple
    occurrences of the same field;

    // Build a field affinity graph for this instance
    for each pair of instance fields
    {
      compute field affinity edge weight;
    }
  }

  //Combine instance field affinity graphs to create a
  // structure field affinity graph
  for each pair of structure fields
  {
    find all structure instances for which
    this pair of fields has an affinity edge
    and compute a weighted affinity;
  }
}
```

Figure 6-7. Processing the structure access database.

interval. Edge weights are proportional to the frequency of contemporaneous access. The field affinities in each instance field affinity graph are weighted by the fraction of total execution frequency of all structure fields accessed through that instance to the total number of structure field accesses thorough all instances. Finally, all weighted instance affinity graphs of each structure type are combined to produce a single affinity graph for each structure (Figure 6-7) by adding together the weighted field affinities for the same pair of fields. Since all instances of a structure type must share the same field order, this use of weights to combine multiple instance affinity graphs into a single structure affinity graph favors field affinities that occur in frequently accessed instances.

6.2.4 Producing Structure Field Orderings

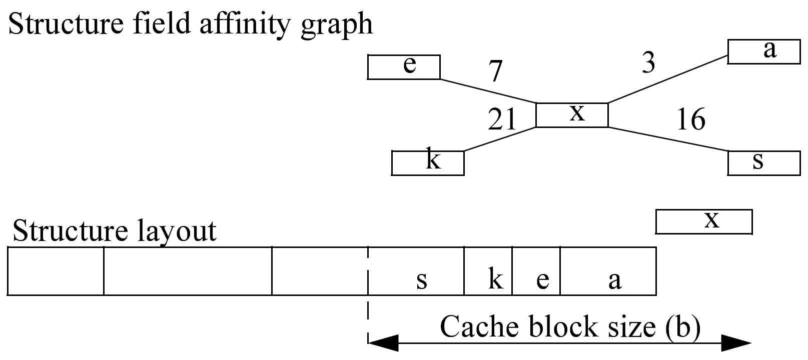
The field reordering problem can be formally stated as follows. Given a structure S, with fields f_1, ... , f_n, of sizes s_1, ..., s_n, respectively, and an access sequence A, which is a sequence of fields that represents the program's field accesses in program-access order, find the optimal order of structure fields, such that the layout minimizes the structure's cache miss rate, if the cache configuration is $<c, b, a>$ (i.e., cache capacity is c sets, cache block size is b bytes, and associativity is a). This problem can be shown to be NP-complete via a reduction similar to the one given by Thabit [74].

Even if it were possible to compute the optimal structure field order efficiently, the solution would be valid only if the program is rerun on the same input data set, and is deterministic. However, we are interested in reordering fields so the resulting program performs well for a wide range of inputs. Since contemporaneous field accesses and field-access frequencies for different program inputs are unpredictable, we resort to using heuristics.

In addition, since structure alignment with respect to cache block boundaries can only be determined at run time (unless the *malloc* pointer is suitably manipulated), our approach is to attempt to increase inherent locality by placing fields with high temporal affinity near each other—so that they are likely to reside in the same cache block—rather than try to pack fields exactly into cache blocks. If alignment (natural boundary) constraints would force a gap in the layout that alternative high temporal affinity fields are unable to occupy, we attempt to fill these with structure fields that were not accessed in the profiling scenario.

We introduce the notion of configuration locality to explain `bbcache`'s algorithm. Configuration locality attempts to capture a layout's inherent locality. The first step is to compute a layout affinity for each field, which is the sum of its weighted affinities with neighboring fields in the layout up to a predefined horizon (presumably equivalent to the cache block size) on either side. If field f_i is surrounded by fields f_1, ..., f_n, in the layout, then its layout affinity is:

$$\textit{Field layout affinity}(f_i) = wt(f_1, f_i) * aff(f_1, f_i) + \ldots$$



$$\Delta(configuration \text{ Ð } locality) = affinity(x, a) \times \frac{b \text{ Ð } 4}{b} + affinity(x, e) \times \frac{b \text{ Ð } 6}{b}$$
$$+ affinity(x, k) \times \frac{b \text{ Ð } 8}{b} + affinity(x, s) \times \frac{b \text{ Ð } 12}{b}$$

Figure 6-8. Producing field orders from the structure field affinity graph.

$$+ wt(f_n, f_i) * aff(f_n, f_i) \qquad \text{EQ 4.}$$

The weights correspond to the distance between the fields—the number of bytes separating the start of the fields—and are a measure of the probability that the fields will end up in the same cache block. The weighting factor used is:

$$wt(f_i, f_j) = (monus^1(cache\_block\_size, dist(f_i, f_j)) / cache\_block\_size)$$

A structure's configuration locality is the sum of its field layout affinities. Figure 6-8 illustrates the process of computing the increase in configuration locality from adding field *x* to an existing layout.

The field reordering algorithm uses a greedy approach to produce a recommended field ordering from a structure field affinity graph. It starts by adding to the layout the pair of fields connected by the maximum affinity edge in the structure field affinity graph. Then at each step, a single field is appended to the existing layout. The field selected is the one

1. where *monus*(a,b) = a – b if a > b, = 0 otherwise.

that increases configuration locality by the largest amount at that point in the computation. This process is repeated until all structure fields are laid out.

We investigated several other algorithms that performed different traversals of the structure field affinity graph (greedy breadth-first, greedy depth-first, depth-first with lookahead), as well as an algorithm that bin-packed fields into cache blocks. The greedy algorithm used performed better than all the alternatives (which produced no performance improvement for Microsoft SQL server 7.0). We believe this occurred for the following reason: Our algorithm attempts to improve a structure's inherent locality, yet is conscious that fields that are far apart are unlikely to reside in the same cache block. The depth-first and breadth-first traversals of the affinity graph do not do as good a job of increasing a structure's inherent locality. In addition, if structures are not aligned on cache block boundaries, the packing approach, which could pack fields with no affinity in consecutive cache blocks, may perform poorly.

6.2.5 Evaluating Structure Field Orderings

While the best way to evaluate a structure field ordering is to measure its impact on performance, this entails a tedious cycle of editing, recompiling, and rerunning the application. A quality metric for structure field orderings can help compare a recommended layout against the original layout and help evaluate alternative layouts, without rerunning the application. This is especially useful when field layout constraints prevent directly following the field ordering recommendations.

bbcache provides two metrics to evaluate structure field orderings, as well as a query facility to compare alternative layouts. The first is a metric of the average number of structure cache blocks active during an application's execution (i.e., a measure of a structure's cache block working set or cache pressure). This metric is computed by combining temporal information for field accesses with a structure's field order to determine active cache blocks. A program's execution is divided into temporal intervals of 100ms each. This metric assumes that structures start on cache block boundaries, and uses the field order (and field sizes) to assign fields to cache blocks. If any of the fields in a cache block are accessed during an execution interval, that block is considered to be active in that interval. Let n represent the total number of program execution intervals, and $b_1, ..., b_n$ the number of active structure cache blocks in each of these intervals. Then a structure's cache block pressure is:

$$\textit{Cache block pressure} = \Sigma(b_1, ...,b_n) / n \qquad \text{EQ 5.}$$

The second metric is a locality metric that measures a structure's average cache block utilization. Let f_{ij} represent the fraction of cache block j accessed (determined by accessed field sizes relative to the cache block size) in program execution interval i, then:

$$\textit{Cache block utilization} = \Sigma(f_{11}, ...,f_{nbn}) / \Sigma(b_1, ...,b_n) \qquad \text{EQ 6.}$$

6.3 Experimental Evaluation of Class Splitting

This section presents experiments that measure the effectiveness of the splitting algorithm and its impact on the performance of Java programs. As described earlier, we used the University of Washington Vortex compiler infrastructure with aggressive optimization (-o2). Table 5.1 describes the Java benchmarks and Section 5.5.1 describes the experimental methodology.

The first set of experiments were designed to investigate the potential for class splitting in the Java benchmarks, study the behavior of our splitting algorithm, and examine the sensitivity of splitting decisions to program inputs.

Table 6.1 shows that the five Java benchmarks for two different sets of inputs have a significant number of classes (17–46% of all accessed classes) that are candidates for splitting (i.e., live and sufficiently large). Even more promising, 26–100% of these candidate classes have field-access profiles that justify splitting the class. The cold fields include variables that handle error conditions, store limit values, and reference auxiliary objects

Program	# of classes (static)	# of accessed classes	# of 'live' classes	# of candidate classes (live & suitably sized)	# of split classes	Splitting success ratio (#split/ #candidates)
cassowary	27	12	6	2	2	100.0%
espresso (i/p 1)	104	72	57	33	11 (8)	33.3%
espresso (i/p 2)	104	69	54	30	9 (8)	30.0%
javac (i/p 1)	169	92	72	25	13 (11)	52.0%
javac (i/p 2)	169	86	68	23	11 (11)	47.8%
javadoc (i/p 1)	173	67	38	13	9 (7)	69.2%
javadoc (i/p 2)	173	62	30	11	7 (7)	63.6%
pizza (i/p 1)	207	100	72	39	10 (9)	25.6%
pizza (i/p 2)	207	95	69	36	10 (9)	27.8%

Table 6.1: Potential for class splitting.

that are not on the critical path for traversing the data structure. The splitting algorithm is fairly insensitive to the input data used for profiling field accesses. For all benchmarks, regardless of input data set, 73–100% of the classes selected for splitting were identical (the second number enclosed in brackets indicates the number of common classes split with different inputs), with the same fields labeled hot or cold, barring a few exceptions. Closer examination of the classes split with one input set and not the other revealed these to be classes with the smallest normalized temperature differentials (though greater than 0.5).

Table 6.2 and Table 6.3 analyze the characteristics of the split classes in more detail. Accesses to fields in split classes account for 45–64% of the total number of program field accesses. The average dynamic sizes of split classes were computed by weighting each split class with the number of its split instances. The splitting algorithm reduces dynamic class sizes by 17–23% (cassowary shows a 68% reduction), and with the exception of jav-

Program	Split class access /total prog. accesses	Avg. normalized temperature differential	Additional space allocated for cold class field ref (bytes)
cassowary	45.8%	98.6%	56
espresso (i/p 1)	55.3%	79.2%	74,464
espresso (i/p 2)	59.4%	79.5%	58,160
javac (i/p 1)	45.4%	75.1%	50,372
javac (i/p 2)	47.1%	79.8%	36,604
javadoc (i/p 1)	56.6%	85.7%	20,880
javadoc (i/p 2)	57.7%	85.2%	12,740
pizza (i/p 1)	58.9%	79.4%	55,652
pizza (i/p 2)	64.0%	82.1%	38,004

Table 6.2: Characteristics of split classes.

Program	Avg. pre-split class size (static)	Avg. pre-split class size (dyn)	Avg. post-split (hot) class size (static)	Avg. post-split (hot) class size (dyn)	Avg. reduction in (hot) class size (static)	Avg. reduction in (hot) class size (dyn)
cassowary	48.0	76.0	18.0	24.0	62.5%	68.4%
espresso (i/p 1)	41.4	44.8	28.3	34.7	31.6%	22.5%
espresso (i/p 2)	42.1	36.2	25.7	30.1	39.0%	16.9%
javac (i/p 1)	45.6	26.3	27.2	21.6	40.4%	17.9%
javac (i/p 2)	49.2	27.2	28.6	22.4	41.9%	17.6%
javadoc (i/p 1)	55.0	48.4	29.3	38.1	46.7%	21.3%
javadoc (i/p 2)	59.4	55.1	33.6	44.0	43.4%	20.1%
pizza (i/p 1)	37.8	34.4	22.9	27.3	39.4%	20.6%
pizza (i/p 2)	39.4	30.9	23.7	24.4	39.9%	21.0%

Table 6.3: Impact of splitting on class size.

Program	L2 cache miss rate (base)	L2 cache miss rate (CL)	L2 cache miss rate (CL + CS)	% reduction in L2 miss rate (CL)	% reduction in L2 miss rate (CL + CS)
cassowary	8.6%	6.1%	5.2%	29.1%	39.5%
espresso	9.8%	8.2%	5.6%	16.3%	42.9%
javac	9.6%	7.7%	6.7%	19.8%	30.2%
javadoc	6.5%	5.3%	4.6%	18.5%	29.2%
pizza	9.0%	7.5%	5.4%	16.7%	40.0%

Table 6.4: Impact of hot/cold object partitioning on L2 miss rate.

adoc, permits two or more hot instances to fit in a cache block. The normalized temperature differentials are high (77–99%), indicating significant disparity between hot and cold field accesses. Finally, the additional space costs for the reference from the hot portion to the cold portion are modest—on the order of 13–74KB—compared with the amount of heap-allocated data.

Next, the UltraSPARC's [72] hardware counters were used to measure the effect of our cache-conscious object layouts on cache miss rates. Each experiment was repeated five times and the average value reported (in all cases the variation between the smallest and largest values was less than 3%). With the exception of cassowary, the test input data set differed from the input data used to generate field-access statistics for class splitting. We measured the impact of our garbage collection scheme for cache-conscious object co-location on the hot/cold split classes versions of the Java programs and compared it against the original versions of the programs. The results are shown in Table 6.4 (we do not report L1 miss rates since L1 cache blocks are only 16 bytes and miss rates were marginally affected, if at all). **CL** represents direct application of our garbage collection scheme for cache-conscious object co-location, and **CL + CS** represents this scheme combined with

Program	Execution time in secs (base)	Execution time in secs (CL)	Execution time in secs (CL + CS)	% reduction in execution time (CL)	% reduction in execution time (CL + CS)
cassowary	34.46	27.67	25.73	19.7	25.3
espresso	44.94	40.67	32.46	9.5	27.8
javac	59.89	53.18	49.14	11.2	17.9
javadoc	44.42	39.26	36.15	11.6	18.6
pizza	28.59	25.78	21.09	9.8	26.2

Table 6.5: Impact of hot/cold object partitioning on execution time.

hot/cold class splitting. The results indicate that the garbage collection scheme for cache-conscious object co-location reduces L2 miss rates by 16–29% and our hot/cold class splitting increases the effectiveness of this scheme, reducing L2 miss rates by a further 10–27%.

Finally, we measured the impact of our techniques on execution time. The results shown in Table 6.5 indicate that hot/cold class splitting also affects execution time, producing improvements of 6–18% over and above the 10–20% gains from the garbage collection scheme for cache-conscious co-location.

6.4 Experimental Evaluation of Field Reordering

We used a 4 processor 400MHz Pentium II Xeon system with a 1MB L2 cache per processor. The system had 4GB of main memory with 200 disks, each a 7200 rpm Clarion fiber channel drive. The system was running Microsoft SQL Server 7.0 on top of Windows NT 4.0. We ran the TPC-C [22] benchmark on this system. Microsoft SQL Server was first instrumented to collect a trace of structure field accesses while running TPC-C. `bbcache` used this trace to produce recommendations for ordering structure fields.

Structur e	Cache block utilization (original order)	Cache block utilization (recommended order)	Cache pressure (original order)	Cache pressure (recommended order)
ExecCxt	0.607	0.711	4.216	3.173
SargMgr	0.714	0.992	1.753	0.876
Pss	0.589	0.643	8.611	5.312
Xdes	0.615	0.738	2.734	1.553
Buf	0.698	0.730	2.165	1.670

Table 6.6: bbcache evaluation metrics for 5 active SQL Server structures.

Out of the almost 2,000 structures defined in the SQL Server source, bbcache indicated that 163 accounted for over 98% of structure accesses for the TPC-C workload. In addition, the top 25 of these 163 active structures account for over 85% of structure accesses. For this reason, we focused on these 25 active structures.

SQL Server uses a number of persistent, on-disk structures that cannot have their fields reordered without affecting compatibility (Section 6.1.4). In addition, there are dependences—e.g., because of the use of casting—between structures that prevent reordering the fields of one without also reordering the fields of the other. It should be noted that SQL server is a highly tuned commercial application, and many of the 25 active structures previously had their fields reordered by hand. We used bbcache to select 5 structures that had no constraints on reordering and which showed the largest potential benefits according to the cost and locality metrics provided (Table 6.6). We reordered these 5 structures according to bbcache's recommendations and ran the TPC-C benchmark on this modified SQL Server several times. The performance of the modified SQL Server was consistently better by 2–3%.

6.5 Related Work

Recent research has focused on reorganizing the data layout of pointer-based codes to improve memory-system performance [12, 19, 20, 76, 39]. Calder et al. apply a compiler-directed approach that uses profile information to place global data, constants, stack variables, and heap objects [12]. Their techniques produced significant improvements for globals and stack data, but only modest gains for heap objects. Their approach differs from ours in two respects. First, they adjusted the placement of entire objects, while we reorganized the internal fields of objects. Second, we focus on heap objects and demonstrate large performance gains.

Chapter 4 describes two tools—a data reorganizer for tree-like structures and a cache-conscious heap allocator—for improving the cache performance of C programs [19]. The tools require few source code modifications and produce significant performance improvements. Both tools reorganize the memory arrangement of entire objects. This work complements that work, since the combination of the two techniques can yield larger benefits than either alone.

Chapter 5 showed how to use generational garbage collection to reorganize data structures so that objects with high temporal affinity are placed near each other, so they are likely to reside in the same cache block [20]. We extend this technique and increase its effectiveness by partitioning classes into a hot and cold portion.

Truong et al. also suggest field reorganization for C structures. They develop a memory-allocation library to support interleaving identical fields of different instances of a structure that are referenced together, and demonstrate significant reductions in cache miss rates and execution times [76]. Our work complements theirs since they perform field reorganization manually using profiling data, whereas we describe a tool—`bbcache`—that automates part of this process. Moreover, we showed in Chapter 5 how to fully automate cache-conscious layout for languages that support copying garbage collection.

Concurrently, Kistler and Franz describe a technique that uses temporal profiling data to reorder structure fields [39]. Their work differs from ours in four ways: First, they use path profiling data to capture temporal relationships. Second, they optimize their layouts for cache-line fill buffer forwarding, a hardware feature supported on the PowerPC, whereas we optimize layouts for inherent locality. Third, their algorithm divides the affinity graph into cache-line sized cliques. A consequence of this technique is that there may be no affinity between fields placed in consecutive cache lines. Without cache-line alignment at allocation time (i.e., by suitably manipulating the *malloc* pointer), the resultant layout may not perform well. Finally, we provide structure-activity rankings and two metrics for evaluating structure field orderings that permit an informed selection of suitable candidates for structure field reordering.

All's well that ends well; still the fine's the crown;
Whate'er the course, the end is the renown.
—Shakespeare, All's Well that Ends Well

Chapter 7

Conclusion

Traditionally, in-core 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 provide an opportunity to achieve significant performance improvements by redesigning data structures to use caches more effectively. This thesis explores design principles, such as clustering, coloring, and compression, for improving the spatial and temporal locality of pointer-based data structures. The resulting cache-conscious data structures show significant performance benefits. In addition, this thesis provides an analytic framework for quantifying the performance improvement of these cache-conscious data structures. This framework can help guide the cache-conscious design process and make it less of an art.

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 thesis has investigated several strategies that facilitate the creation of cache-conscious pointer structure layouts.

Ideally, the compiler or run-time system should automatically produce cache-conscious pointer structure layouts with no programmer assistance or intervention. Unfortunately, many popular programming languages, such as C and C++, contain low-level language features that make this goal impossible to attain without hardware support. Not surprisingly, this thesis offers no silver bullet for producing cache-conscious structure layouts in these languages. However, it explores a wide variety of techniques that greatly reduce the programming effort and application knowledge required to improve cache performance. These techniques produce a cache-conscious arrangement of structure instances in memory. Additional techniques manipulate the internal organization of fields in a structure instance to make the layout cache-conscious. Finally, the thesis shows that these cache-conscious techniques can be packaged into easy-to-use tools.

This thesis describes a more attractive alternative for languages that support garbage collection. A generational garbage collector can easily be modified to produce cache-conscious data layouts of small objects. The thesis demonstrates the feasibility of low-overhead, real-time profiling of data-access patterns for object-oriented languages and describes a new copying algorithm that uses this information to produce cache-conscious object layouts. Measurements show that this technique reduces cache miss rates and improves program performance significantly. Techniques such as these 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 or C++.

The main contributions of this thesis are:

- Design principles for cache-conscious structure design. While the ideas explored in Chapter 2 are not fundamentally new, they have not been systematically applied to construct cache-conscious pointer structures. In addition, the analytic framework presented in Chapter 3 quantifies the performance benefits of these design principles with a new data structure-centric cache model, and helps make the cache-conscious structure design process less of an art.
- Techniques for making the arrangement of structure instances in memory cache-conscious. The approaches—*cache-conscious allocation* and *cache-conscious reorganization*—presented in Chapter 4 are quite general and even apply to low-level languages, such as C and C++.
- A technique for using a generational garbage collector to implement cache-conscious data placement. Previous work that used garbage collection to improve locality focused on the page level, whereas this research focuses on the cache level. This distinction is important since Chapter 5 demonstrates that improving page locality does not necessarily improve cache locality. In addition, the garbage collection scheme for cache-conscious co-location is completely automatic and requires no programmer assistance or intervention.
- Techniques for making the internal organization of fields in a data structure cache-conscious. While the foregoing techniques, which concentrate on arranging distinct structure instances, work best for structures smaller than half a cache block, the *cache-conscious definition* techniques in Chapter 6 improve the cache behavior of larger structures.

7.1 Future Work

This thesis lays the foundation for the principled design and implementation of cache-conscious data structures. However, there are several issues that need further exploration.

- Can static program analyses provide sufficient information about dynamic structure accesses that would eliminate or reduce the need for profiling?
- Can programmer annotations help the compiler or run-time system in producing cache-conscious structure layouts? If so, what types of annotations would be most useful? [32] contains some possible annotations.
- Are there additional techniques for producing cache-conscious data structures and can they be automated?
- How do other latency reducing techniques, such as prefetching, interact with cache-conscious data structures? [62] contains a study of prefetching cache-conscious lists.
- What kind of hardware support would be most appropriate for this problem? One possible approach is proposed in [49].
- Future work on cache-conscious data structures and algorithms is likely to be very profitable. Will this require a radically new framework for algorithm design?

7.2 Some Final Remarks

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 thesis, an essential aspect of the process of achieving the highest performance on current and future machines.

Bibliography

[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] S. Bederman. Cache Management System using Virtual and Real Tags in the Cache Directory. *IBM Technical Disclosure Bulletin*, 21(11), 1979.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Craig Chambers. Object-Oriented Multi-methods in Cecil. In *Proceedings ECOOP'92, LNCS 615, Springer-Verlag*, pages 33–56, June 1992.

[16] Craig Chambers. The Cecil Language: Specification and Rationale. In *University of Washington Seattle, Technical Report TR-93-03-05*, Mar. 1993.

[17] Craig Chambers, Jeffrey Dean, and David Grove. Whole-Program Optimization of Object-Oriented Languages. In *University of Washington Seattle, Technical Report 96-06-02*, June 1996.

[18] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[19] 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.

[20] 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.

[21] Douglas Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[22] Transaction Processing Council. TPC Benchmark C, Standard Specification, Rev. 3.6.2. June 1997.

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

[24] Roger F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Oct. 1997.

[25] C. Ding and K. Kennedy. Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time. In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI)*, pages 229–241, May 1999.

[26] 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.

[27] Robert Fenichel and Jerome Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, 1969.

[28] 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.

[29] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure Placement Using Temporal Ordering Information. In *Proceedings of MICRO-30*, Dec. 1997.

[30] James Gosling, Bill Joy, and Guy L. Steele Jr. *The Java Language Specification*. Addison-Wesley, 1996.

[31] 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.

[32] L. Hendren, J. Hummell, and A. Nicolau. Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative Programs. In *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 249–260, Jun. 1992.

[33] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[34] Mark D. Hill. A Case for Direct-Mapped Caches. *IEEE Computer*, 21(12):25–40, Dec. 1988.

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

[36] Richard Hudson, Eliot Moss, Amer Diwan, and Christopher Weight. A Language-Independent Garbage Collector Toolkit. In *University of Massachusetts at Amherst technical report TR 91-47*, Sept. 1991.

[37] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley, 1996.

[38] R. E. Kessler and Mark D. Hill. Page Placement Algorithms for Large Real-Index Caches. *ACM Transactions on Computer Systems*, 10(4):338–359, 1992.

[39] T. Kistler and M. Franz. The Case for Dynamic Optimization: Improving Memory-Hierarchy Performance by Continuously Adapting the Internal Storage Layout of Heap Objects at Run-Time. In *Department of Information and Computer Science, University of California at Irvine, Technical Report 99-21*, May 1999.

[40] C. Krintz, B. Calder, H. B. Lee, and B. G. Zorn. Overlapping Execution with Transfer using Non-strict Execution for Mobile Programs. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 159–169, Oct. 1998.

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

[42] 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.

[43] 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.

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

[45] 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.

[46] 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.

[47] H. B. Lee and B. G. Zorn. BIT: A Tool for Instrumenting Java Bytecodes. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems (USITS'97)*, pages 73–83, Dec. 1997.

[48] Henry Lieberman and Carl Hewitt. A Real-Time Garbage Collector Based on Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429, 1983.

[49] C. K. Luk and T. C. Mowry. Memory Forwarding: Enabling Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation. In *The 26th Annual International Symposium on Computer Architecture*, pages 88–99, May 1999.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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*.

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

[60] 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.

[61] 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.

[62] Shai Rubin, David Bernstein, and Michael Rodeh. Virtual Cache Line: A New Technique to Improve Cache Exploitation for Recursive Data Structures. In *Eight International Conference on Compiler Construction, LNCS 1575*, pages 259–273, Mar. 1999.

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

[64] Matthew L. Seidl and Benjamin G. Zorn. Segregating Heap Objects by Reference Behavior and Lifetime. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 12–23, Oct. 1998.

[65] 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.

[66] 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.

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

[68] Burton J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *Real-Time Signal Processing IV*, pages 241–248, 1981.

[69] Michael D. Smith, Mark Horwitz, and Monica S. Lam. Efficient Superscalar Performance Through Boosting. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 248–259, October 1992.

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

[71] 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.

[72] Sun Microelectronics. *UltraSPARC User's Manual*, 1996.

[73] Robert E. Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[74] K. O. Thabit. Cache Management by the Compiler. In *Ph.D. Thesis, Department of Computer Science, Rice University*, 1981.

[75] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1), 1967.

[76] Dan N. Truong, Francois Bodin, and Andre Seznec. Improving cache behavior of dynamically allocated data structures. In *International Conference on Parallel Architectures and Compilation Techniques*, Oct. 1998.

[77] 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.

[78] David Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Apr. 1984.

[79] David Ungar and Frank Jackson. An Adaptive Tenuring Policy for Generation Scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, January 1992.

[80] G. J. Ward. The Radiance Lighting Simulation and Rendering System. In *Proceedings of SIGGRAPH '94*, July 1994.

[81] J. L. White. Address/memory management for a gigantic LISP environment, or, GC considered harmful. In *Conference Record of the 1980 LISP Conference*, pages 119–127, 1980.

[82] 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.*

[83] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collectors. In *1992 ACM Symposium on Lisp and Functional Programming*, pages 32–42, June 1992.

[84] 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.*