Appears in *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, May 1999.

Cache-Conscious Structure Layout

Trishul M. Chilimbi
Computer Sciences Department
University of Wisconsin
1210 West Dayton St.
Madison, WI 53706
chilimbi@cs.wisc.edu

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

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

ABSTRACT

Hardware trends have produced an increasing disparity between processor speeds and memory access times. While a variety of techniques for tolerating or reducing memory latency have been proposed, these are rarely successful for pointer-manipulating programs.

This paper explores a complementary approach that attacks the source (poor reference locality) of the problem rather than its manifestation (memory latency). It demonstrates that careful data organization and layout provides an essential mechanism to improve the cache locality of pointer-manipulating programs and consequently, their performance. It explores two placement techniques—*clustering* and *coloring*—that improve cache performance by increasing a pointer structure's spatial and temporal locality, and by reducing cache-conflicts.

To reduce the cost of applying these techniques, this paper discusses two strategies—*cache-conscious reorganization* and *cache-conscious allocation*—and describes two semi-automatic tools—`ccmorph` and `ccmalloc`—that use these strategies to produce cache-conscious pointer structure layouts. `ccmorph` is a transparent tree reorganizer that utilizes topology information to cluster and color the structure. `ccmalloc` is a cache-conscious heap allocator that attempts to co-locate contemporaneously accessed data elements in the same physical cache block. Our evaluations, with microbenchmarks, several small benchmarks, and a couple of large real-world applications, demonstrate that the cache-conscious structure layouts produced by `ccmorph` and `ccmalloc` offer large performance benefits—in most cases, significantly outperforming state-of-the-art prefetching.

Keywords

Cache-conscious data placement, clustering, coloring, cache-conscious allocation, cache-conscious reorganization

1. INTRODUCTION

The speed of microprocessors has increased 60% per year for almost two decades. Yet, over the same period, the time to access main memory only decreased at 10% per year [32]. The unfortunate, but inevitable, consequence of these trends is a large, and ever-increasing, processor-memory gap. Until recently, memory caches have been the ubiquitous response to this problem [50, 43]. In the beginning, a single cache sufficed, but the increasing gap (now almost two orders of magnitude) requires a hierarchy of caches, which introduces further disparities in memory-access costs.

Many hardware and software techniques—such as prefetching [29, 9, 26, 38], multithreading [25, 44], non-blocking caches [20], dynamic instruction scheduling, and speculative execution—try to reduce or tolerate memory latency. Even so, many programs' performance is dominated by memory references. Moreover, high and variable memory access costs undercut the fundamental random-access memory (RAM) model that most programmers use to understand and design data structures and algorithms.

Over the same period, application workloads have also changed. Predominately scientific applications have broadened into a richer workload. With this shift came a change in data structure, from arrays to a richer mix of pointer-based structures. Not surprisingly, techniques for reducing or tolerating memory latency in scientific applications are often ineffective for pointer manipulating programs [7, 33]. In addition, many techniques are fundamentally limited by their focus on the manifestation of the problem (memory latency), rather than its cause (poor reference locality).

In general, 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 [52, 10, 16]. Two properties of array structures are essential to this work: uniform, random access to elements, and a number theoretic basis for statically analyzing data dependencies. These properties allow compilers to analyze array accesses completely and to reorder them in a way that increases cache locality (loop transformations) without affecting a program's result.

Unfortunately, pointer structures share neither property. However, they possess another, extremely powerful property of *locational transparency*: 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 the mechanism to improve the cache locality of pointer manipulating programs and, consequently, their performance. This paper describes and provides an analytic framework for two placement techniques—*clustering* and *coloring*—that improve cache performance on uniprocessor systems by increasing a data structure's spatial and temporal locality, and reducing cache conflicts.

Applying these techniques may require detailed knowledge of a program's code and data structures, architectural familiarity, and considerable programmer effort. To reduce this cost, we discuss two strategies—*cache-conscious reorganization*, and *cache-conscious allocation*—for applying these placement techniques to produce cache-conscious pointer structure layouts, and describe two semi-automatic tools—`ccmorph` and `ccmalloc`—that embody these strategies. Measurements demonstrate that cache-conscious data layouts produced by `ccmorph` and `ccmalloc` offer large performance benefits—in most cases, significantly outperforming

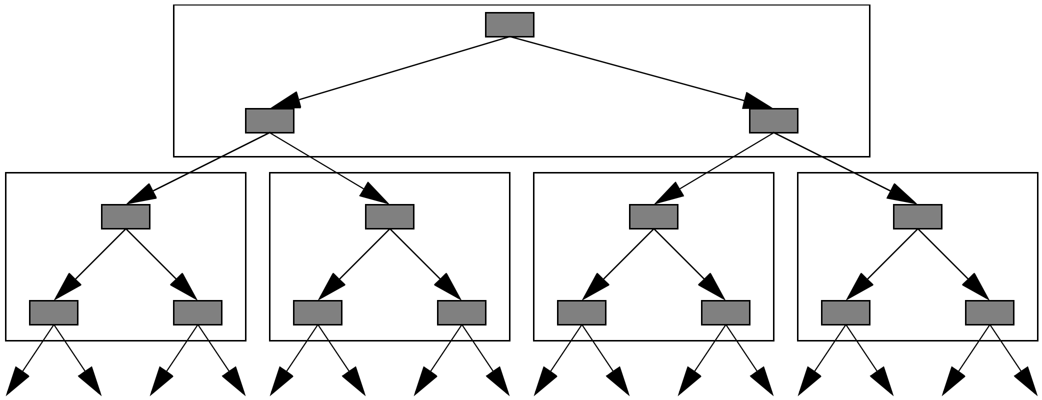

Figure 1. Subtree clustering.

state-of-the-art prefetching.

This paper makes the following contributions:

- ***Cache-conscious data placement techniques.*** Section 2 shows how clustering and coloring can improve a pointer structure's cache performance. 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.
- ***Strategies for applying cache-conscious data placement techniques.*** Section 3 describes two strategies—cache-conscious reorganization and cache-conscious allocation—for applying placement techniques to produce cache-conscious data layouts. Cache-conscious reorganization utilizes structure topology or profile information about data access patterns to transform pointer structure layouts. This approach is incorporated in `ccmorph`, a utility that reorganizes tree-like structures, such as trees, lists, and chained hash tables, by clustering and coloring the structure. A programmer need only supply a function that helps traverse the data structure. Cache-conscious allocation improves on conventional heap allocators by attempting to co-locate contemporaneously accessed data elements in the same physical cache block. The section describes `ccmalloc`, a memory allocator that implements this strategy. In this case, a programmer only must specify an additional argument to `malloc`—a pointer to a structure element likely to be in contemporaneous use.
- ***Evaluation of cache-conscious data placement.*** Section 4 demonstrates the performance benefits of cache-conscious data placement. In microbenchmarks, cache-conscious trees outperform their naive counterparts by a factor of 4–5, and even outperform B-trees by a factor of 1.5. For some pointer-intensive programs in the Olden benchmark suite [36], semi-automatic cache-conscious data placement improves performance 28–194%, and even outperformed state-of-the-art prefetching by 3%–194%. We applied the techniques to full application programs. RADIANCE [49], a widely used ray-tracing program, showed a 42% speedup, and VIS [6], a model verification package, improved by 27%. Significantly, applying `ccmalloc` to the 160,000 line VIS, required little application understanding, and took only a few hours.
- ***Analytic framework.*** Section 5 presents an analytic framework that quantifies the performance benefits of cache-conscious pointer-based data structures. A key part of this framework is a data structure-centric cache model of a series of accesses that traverse a pointer-based data structure. The model characterizes the performance of a pointer-based data structure by its amortized miss rate over a sequence of pointer-path accesses. This paper applies the framework to cache-conscious trees and validates its predictions with a microbenchmark.

2. CACHE-CONSCIOUS DATA PLACEMENT TECHNIQUES

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

2.1 Clustering

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.

An effective way to cluster a tree is to pack subtrees into a cache block. Figure 1 illustrates subtree clustering for a binary tree. An intuitive justification for binary subtree clustering is as follows (detailed analysis is in Section 5.3). For a series of random tree searches, the probability of accessing either child of a node is 1/2. With k nodes in a subtree clustered in a cache block, the expected number of accesses to the block is the height of the subtree, $log_2(k+1)$, which is 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:

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

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.

2.2 Coloring

Caches have finite associativity, which means that only a limited number of concurrently accessed data elements can map to the same cache block without incurring conflict misses. Coloring maps contemporaneously-accessed elements to non-conflicting regions of the cache. Figure 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 region 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. For a tree, the most heavily accessed elements are the nodes near the root of the tree.

3. STRATEGIES FOR APPLYING CACHE-CONSCIOUS DATA PLACEMENT

Designing cache-conscious data structures requires detailed knowledge of a program's code and data structures and considerable programming effort. This section explores two strategies—*cache-conscious reorganization* and *cache-conscious allocation*—for applying placement techniques to produce cache-conscious data layouts, and describes two semi-automatic tools—`ccmorph` and `ccmalloc`—that implement these strategies. Both significantly reduce the level of programming effort, knowledge, and architectural familiarity.

3.1 Cache-Conscious Data Reorganization

A data structure is typically allocated memory 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 specializing 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 [8, 11]. However, a very important class of structures (trees) possess topological properties that permit cache-conscious data reorganization without profiling. This section presents a transparent (semantic-preserving) cache-conscious tree reorganizer (`ccmorph`) that applies the clustering and coloring techniques described in the previous section.

3.1.1 ccmorph

In a language, such as C, with unrestricted pointers, analytical techniques cannot 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 [11]). However, if a programmer guarantees the safety of the operation, `ccmorph` transparently reorganizes a data structure to improve locality by applying the clustering and coloring techniques from Section 2.1 and Section 2.2. Reorganization is appropriate for read-mostly data structures, which are built early in a computation and subsequently heavily referenced. With this approach, neither the construction or 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.

`ccmorph` operates on tree-like structures with homogeneous elements and without external pointers into the middle of the structure (or any data structure that can be decomposed into components satisfying this property). However, it allows a liberal definition of a tree in which elements may contain a parent or predecessor pointer. A programmer supplies `ccmorph` (which is templatized with respect to the structure type) with a pointer to the root of a data structure, a function to traverse the structure, and cache parameters. For example, Figure 3 contains the code used to reorganize the quadtree data structure in the Olden benchmark *perimeter* with the programmer supplying the *next_node* function.

`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 (Figure 1). 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 (Figure 2). `ccmorph` determines the values of p and size of subtrees from the cache parameters and 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 5.

3.2 Cache-Conscious Heap Allocation

Although `ccmorph` requires little programming effort, it currently only works for tree-like structures that can be moved. In addition, incorrect usage of `ccmorph` can affect program correctness. A

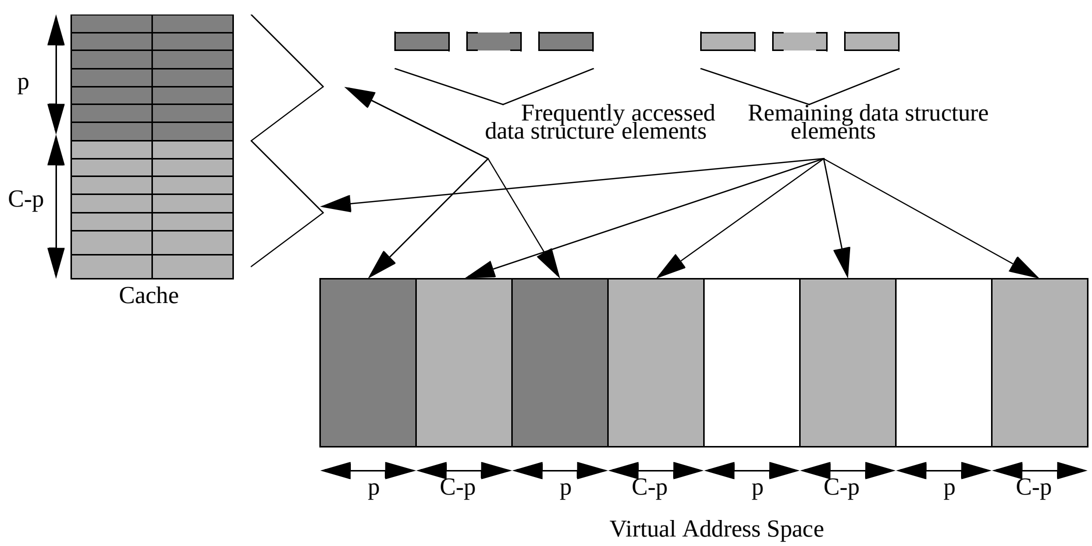

Figure 2. Coloring data structure elements to reduce cache conflicts.

```
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 3. `ccmorph`: Transparent cache-conscious data reorganization.

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, and not correctness.

3.2.1 ccmalloc

`ccmalloc` is a memory allocator similar to `malloc`, which takes an additional parameter that points to an existing data structure element likely to be accessed contemporaneously (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 contains code from the Olden benchmark *health*, which illustrates the approach. Our experience with `ccmalloc` indicates that a programmer unfamiliar with an application can select a suitable parameter by local examination of code surrounding the allocation statement and obtain good results (see Section 5).

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.

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

An important issue is where to allocate a new data item if a cache block is full. `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, and improve 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. EVALUATION OF CACHE-CONSCIOUS DATA PLACEMENT

To evaluate our cache-conscious placement techniques, we use a combination of a microbenchmark, and 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 microbenchmark performed a large number of random searches on different types of balanced trees. 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.

4.1 Methodology

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

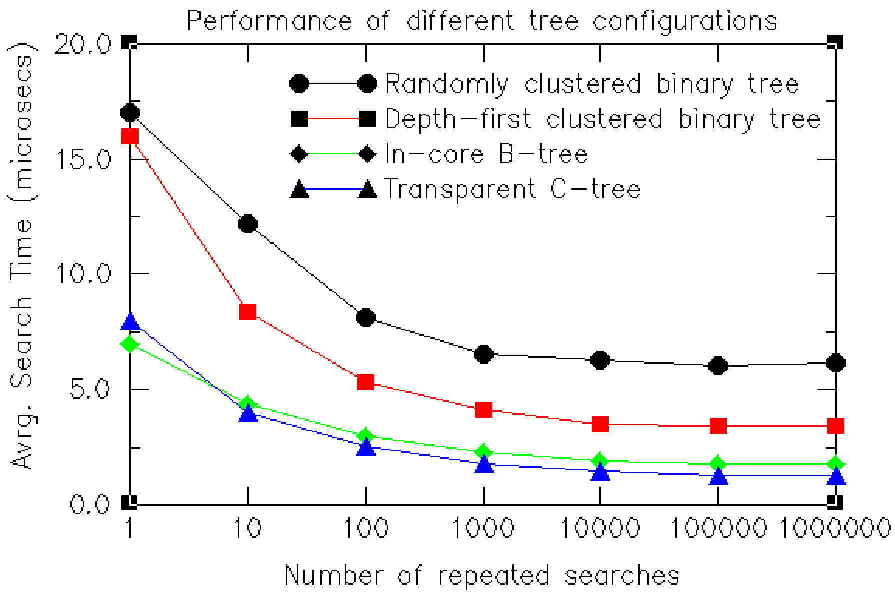

Figure 5. Binary tree microbenchmark.

4.2 Tree Microbenchmark

This microbenchmark measures the performance of `ccmorph` on a large binary search tree, which we call a transparent C-tree. We compare its performance against an in-core B-tree, also colored to reduce cache conflicts, and against random and depth-first clustered binary trees. The microbenchmark does not perform insertions or deletions. The tree contained 2,097,151 keys and consumes 40 MB of memory (forty times the L2 cache's size). Since the L1 cache block size is 16 bytes and its capacity is 16K bytes, it provides practically no clustering or reuse, and hence its miss rate was very close to one. We measured the average search time for a randomly selected element, while varying the number of repeated searches to 1 million. Figure 5 shows that both B-trees and transparent C-trees outperform randomly clustered binary trees by up to a factor of 4–5, and depth-first clustered binary trees by up to a factor of 2.5–3. Moreover, transparent C-trees outperform B-trees by a factor of 1.5. The reason for this is that B-trees reserve extra space in tree nodes to handle insertion gracefully, and hence do not manage cache space as efficiently as transparent C-trees. However, we expect B-trees to perform better than transparent C-trees when trees change due to insertions and deletions.

4.3 Macrobenchmarks

We also studied the impact of cache-conscious data placement on two real-world applications. RADIANCE is a tool for modeling the distribution of visible radiation in an illuminated space [49]. Its input is a three-dimensional geometric model of the space. Using radiosity equations and ray tracing, it produces a map of spectral radiance values in a color image. RADIANCE's primary data structure is an octree, which represents the scene to be modeled. This structure is highly optimized. The program uses implicit knowledge of the structure's layout to eliminate pointers, much like an implicit heap, and it lays out this structure in depth-first order (consequently, it did not make sense to use `ccmalloc` in

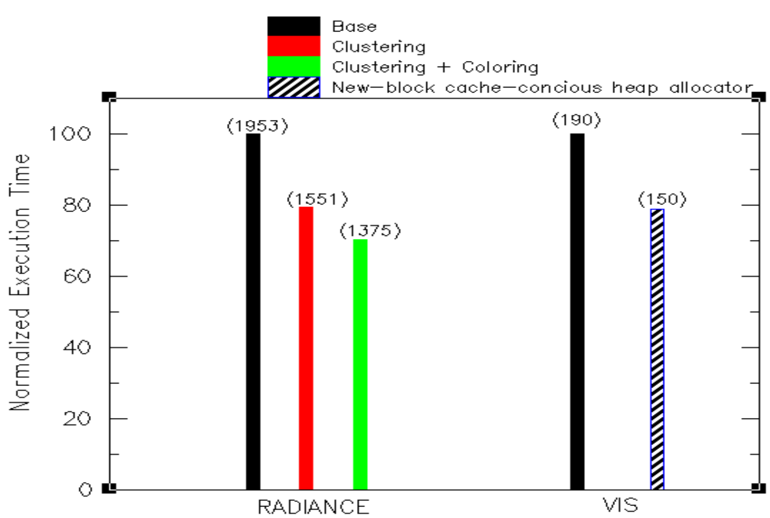

Figure 6. RADIANCE and VIS Applications. Actual execution times above each bar.

Table 1: Simulation Parameters.

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

this case). We changed the octree to use subtree clustering and colored the data structure to reduce cache conflicts. The performance results includes the overhead of restructuring the octree.

VIS (Verification Interacting with Synthesis) is a system for formal verification, synthesis, and simulation of finite state systems [6]. VIS synthesizes finite state systems and/or verifies properties of these systems from Verilog descriptions. The fundamental data structure used in VIS is a multi-level network of latches and combinational gates, which is represented by Binary Decision Diagrams (BDDs). Since BDDs are DAGs, `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 4.4).

Figure 6 shows the results. Cache-conscious clustering and coloring produced a speedup of 42% for RADIANCE, and cache-conscious heap allocation resulted in a speedup of 27% for VIS. The result for VIS demonstrates that cache-conscious data placement can even improve the performance of graph-like data structures, in which data elements have multiple parents. Significantly, very few changes to these 60–160 thousand line programs produced large performance improvements. In addition, the modifications to VIS were accomplished in a few hours, with little understanding of the application.

4.4 Olden Benchmarks

We performed detailed, cycle-by-cycle uniprocessor simulations of the four Olden benchmarks using RSIM [30]. RSIM is an execution driven simulator that models a dynamically-scheduled, out-of-order processor similar 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 1 contains the simulation parameters.

Table 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 hardware prefetching (prefetching all loads and stores currently in the reorder buffer) and software prefetching (we implement Luk and Mowry's greedy prefetching scheme [26] by hand).

Figure 7 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 [31, 37]. 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 and are competitive with software prefetching for *treeadd*, and outperform both

Table 2: Benchmark characteristics.

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

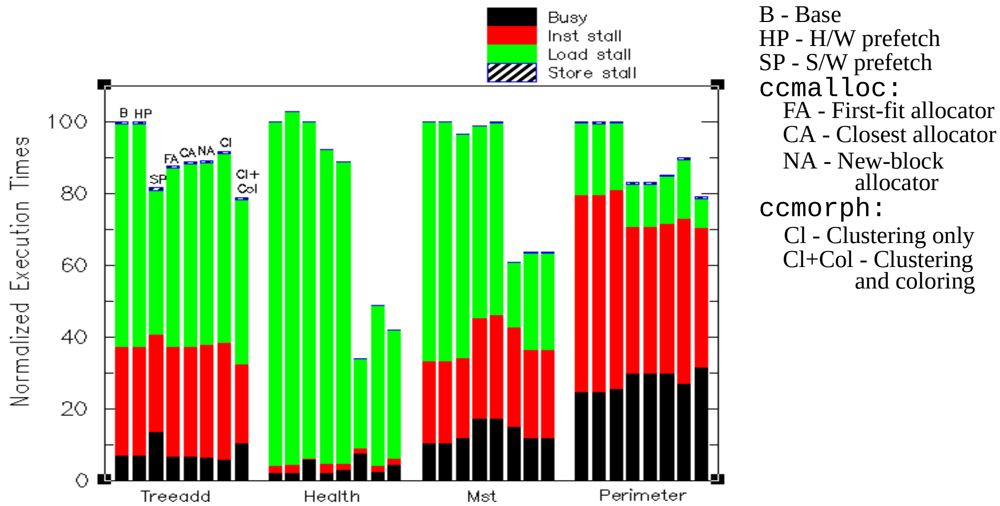

Figure 7. Performance of cache-conscious data placement.

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

Health's primary data structure is 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 collisions. 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.

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`.

4.5 Discussion

Table 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 paper focus on single data structures. Real programs, of course, use multiple data structures, though often references to one structure predominates. Our techniques can be applied to each structure in turn to improve its performance. Future work will consider interactions among different structures.

Our cache-conscious structure layout techniques place contemporaneously accessed elements in the same cache block. While this will always improve uniprocessor cache performance, 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.

5. ANALYTIC FRAMEWORK

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

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

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

vides 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 paper applies the analytic framework to predict the steady-state performance of cache-conscious trees.

5.1 Analytic Model

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

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

t_h: level 1 cache access time

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

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

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

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

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

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

We examine this amortized miss rate for a cache configuration $C<c, b, a>$, where c is the cache capacity in sets, b is the cache block size in words, and a is the cache associativity. Consider a pointer-based data structure consisting of n homogenous elements, subjected to a random sequence of pointer-path accesses of the same type. Let D be a pointer-path access function that represents the average number of unique references required to access an element of the structure. D depends on the data structure, and the type of pointer-path access (if the pointer-path accesses are not of the same type, D additionally depends on the distribution of the different access types). For example, D is $\log_2(n+1)$ for a key search on a balanced binary search tree. Let the size of an individual structure element be e. If $e < b$, then $\lfloor b/e \rfloor$ is the number of structure elements that fit in a cache block. Let 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

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

Let R represent the number of elements of the data structure required for the current pointer-path access that are already present in the cache because of prior accesses. $R(i)$ 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 it follows that

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

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

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

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

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), 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, the amortized steady-state miss rate m_s of a data structure can be approximated by its amortized miss rate $m_a(p)$, for a large, random sequence of pointer-path accesses p, all of the same type, as follows

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

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.

5.2 Speedup Analysis

We use the model to derive an equation for speedup in terms of

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

Figure 8. Cache-conscious speedup.

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

Figure 9. Cache-conscious binary tree.

cache miss rates 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 $= (t_{memory})_{Naive} / (t_{memory})_{Cache\text{-}conscious}$

When only the structure layout is changed, the number of memory references remains the same and the equation reduces to that in Figure 8.

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 $(m_{L1})_{Naive} = (m_{L2})_{Naive} = 1$.

5.3 Steady-State Performance Analysis

This section demonstrates how to calculate the steady-state performance of a cache-conscious tree (see Section 4.2) subjected to a series of random key searches.

Consider a balanced, complete binary tree of n nodes. Let the size of a node be e words. If the cache block size is b words and $e < b$, up to $\lfloor b/e \rfloor$ nodes can be clustered in a cache block. Let subtrees of size $k = \lfloor b/e \rfloor$ nodes fit in a cache block. The tree is colored so the top $(c/2 \times \lfloor b/e \rfloor \times a)$ nodes of the tree map uniquely to the first $c/2$ sets of the cache with no conflicts and the remaining nodes of the tree map into next $c/2$ sets of the cache (other divisions of the cache are possible).

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 large, we can ignore the start-up behavior, and approximate the data structure's performance by its amortized steady-state miss rate as shown in Figure 9.

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

5.4 Model Validation

This section validates the model's predictions of performance improvement. The experimental setup is the same as before (see Section 4.1). The tree microbenchmark is used for the experiments of 1 million repeated searches for randomly generated keys in a tree (Section 4.2). We apply the model to predict the performance advantage of transparent C-trees, which use both subtree clustering and coloring, over their naive counterpart. 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 10. As the graph shows, the model has good predictive power, underestimating the actual speedup by only 15% and accurately predicting the shape of the curve. 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.

6. RELATED WORK

Previous research has attacked the processor-memory gap by reordering computations to increase spatial and temporal locality [16, 52, 10]. Most of this work focused on regular array accesses. Gan-

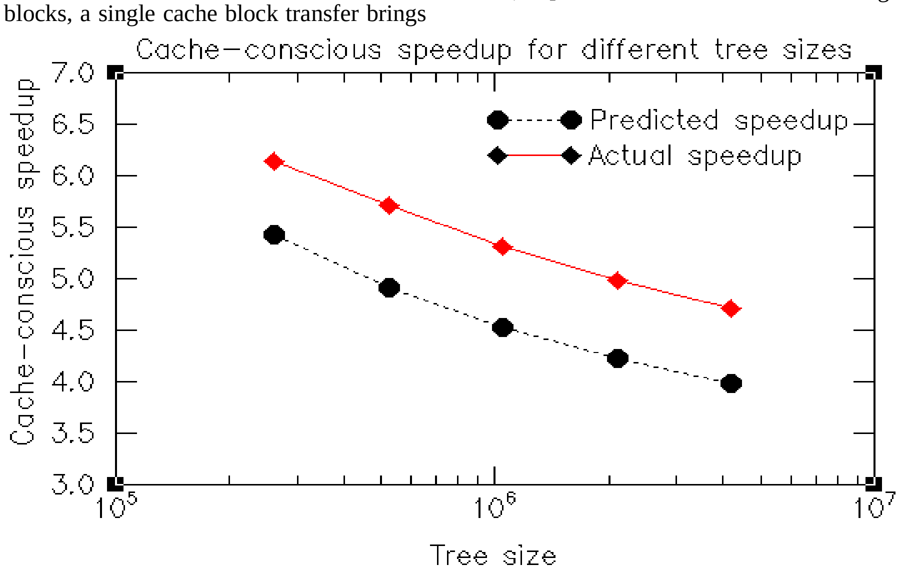

Figure 10. Predicted and actual speedup for C-trees.

non et al. [16] studied an exhaustive approach that generated all permutations of a loop nest and selected the best one using an evaluation function. Wolf and Lam [52] developed a loop transformation theory, based on unimodular matrix transformations, and used a heuristic to select the best combination of loop transformations. Carr et. al [10] used a simple model of spatial and temporal reuse of cache lines to select compound loop transformations. 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 is impossible in general.

Database researchers long ago faced a similar performance gap between main memory and disk speeds. They designed specialized data structures, such as B-trees [4, 13], to bridge this gap. In addition, databases use clustering [3, 48, 15, 5] and compression [13] to improve virtual memory performance. Clustering has also been used to improve virtual memory performance of Smalltalk and LISP systems [28, 46, 51, 21, 14] by reorganizing data structures during garbage collection.

Seidl and Zorn [40] 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. 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.

Recently Chilimbi and Larus [11] used a generational garbage collector to implement cache-conscious data placement. They collect low-overhead, real-time profiling information about data access patterns and applied a new copying algorithm that uses this information to produce a cache-conscious object layout. That work relies on properties of object-oriented programs and requires copying garbage collection, whereas this paper focuses on C programs.

A more recent paper by Chilimbi et al. [12] describes two techniques—structure splitting and field reorganization—for cache-conscious structure definition, and demonstrates performance improvements for C and Java. Truong et al. [47] also suggest field reorganization for C structures. These works complement this one, as they are concerned with improving the cache performance of a data structure by reorganizing its internal layout, while the orthogonal techniques in this paper improve performance by arranging collections of structures.

Calder et al. [8] applied placement techniques developed for instruction caches [17, 34, 27] to data. 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 gains for stack objects and globals. By contrast, we provide tools for cache-conscious heap layout 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.

Researchers have also used empirical models of program behavior [2, 39, 45] to analyze cache performance [35, 42, 18]. 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 [41]. Agarwal's model uses a large number of parameters, some of which appear to require measurements to calibrate. He provides performance validation that shows that the model's predictions are quite accurate. However, the model's complexity and large number of parameters, makes it difficult to gain insight into the impact of different cache parameters on performance. Singh presents a technique for calculating the cache miss rate for fully associative caches from a mathematical model of 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 [19] proposed the simple 3C model, which classifies cache misses into three categories—compulsory, capacity, and conflict. The model provides an intuitive explanation for the causes of cache misses, but it lacks predictive power. These models focus on analyzing and predicting a program's cache performance, while we focus on the cache performance of individual in-core pointer structures.

Lam et al. [22] developed a theoretical model of data conflicts in the cache and analyzed the implications for blocked array algorithms. They showed that cache interference is highly sensitive to the stride of data accesses and the size of blocks, which can result in wide variation in performance for different matrix sizes. Their cache model captures loop nests that access arrays in a regular manner, while our model focuses on series of pointer-path accesses to in-core pointer-based data structures.

LaMarca and Ladner [23, 24] explored the interaction of caches and sorting algorithms. 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" [2], and is algorithm-centric, whereas ours is data structure-centric, and specifically targets correlations between multiple accesses to the same data structure.

7. CONCLUSIONS

Traditionally, in-core pointer-based data structures were designed and programmed as if memory access costs were uniform. Increasingly expensive memory hierarchies open an opportunity to achieve significant performance improvements by redesigning data structures to use caches more effectively. While techniques such as clustering, and coloring can improve the spatial and temporal locality of pointer-based data structures, applying them to existing codes may require considerable effort. This paper shows that cache-conscious techniques can be packaged into easy-to-use tools. Our structure reorganizer, `ccmorph`, and cache-conscious memory allocator, `ccmalloc`, greatly reduce the programming effort and application knowledge required to improve cache performance.

While the cache-conscious structure layout tools described in this paper are fairly automated, they still require programmer assistance to identify tree-structures that can be moved, and suitable candidates for cache block co-location. Future work can explore two directions to reduce the amount of programmer effort: static program analyses and profiling. Finally, we believe that compilers and run-time systems can help close the processor-memory performance gap.

8. ACKNOWLEDGEMENTS

The authors would like to thank Thomas Ball, Brad Calder, Bob Fitzgerald, Anoop Gupta, Manoj Plakal, Thomas Reps, and the anonymous referees for their useful comments. This research is supported by NSF NYI Award CCR-9357779, with support from Sun Microsystems, and NSF Grant MIP-9625558.

9. REFERENCES

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

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

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

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

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

[6] 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.

[7] 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.

[8] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. "Cache-conscious data placement." In *Proceedings of the Eight International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 139-149, Oct. 1998.

[9] 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.

[10] 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.

[11] Trishul M. Chilimbi, and James R. Larus. "Using generational garbage collection to implement cache-conscious data placement." In *Proceedings of the 1998 International Symposium on Memory Management*, Oct. 1998.

[12] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. "Cache-conscious structure definition." In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, May 1999.

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

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

[15] 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.

[16] 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.

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

[18] 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.

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

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

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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*.

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

[36] 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.

[37] 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.

[38] A. Roth, A. Moshovos, and G.S. Sohi. "Dependence based prefetching for linked data structures." In *Proceedings of the Eight International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 115–126, Oct. 1998.

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

[40] M. L. Seidl, and B. G. Zorn "Segregating heap objects by reference behavior and lifetime." In *Proceedings of the Eight International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 12-23, Oct. 1998.

[41] 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.

[42] 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.

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

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

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

[46] 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.

[47] 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.

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

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

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

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

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