Cache-Conscious Data Placement

Brad Calder Chandra Krintz Simmi John

Dept. of Computer Science and Engineering
University of California, San Diego
{calder,ckrintz,sjohn}@cs.ucsd.edu

Todd Austin

Microcomputer Research Labs
Intel Corporation
taustin@ichips.intel.com

Abstract

As the gap between memory and processor speeds continues to widen, cache efficiency is an increasingly important component of processor performance. Compiler techniques have been used to improve instruction cache performance by mapping code with temporal locality to different cache blocks in the virtual address space eliminating cache conflicts. These code placement techniques can be applied directly to the problem of placing data for improved data cache performance.

In this paper we present a general framework for Cache Conscious Data Placement. *This is 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. The placement of data objects is guided by a temporal relationship graph between objects generated via profiling. Our results show that profile driven data placement significantly reduces the data miss rate by 24% on average.*

1 Introduction

Much effort has been invested in reducing the impact of cache misses on program performance. As with any other latency, cache miss latency can be tolerated using compile-time techniques such as instruction scheduling [19], or run-time techniques including out-of-order issue, decoupled execution [31], or non-blocking loads [9]. It is also possible to reduce the latency of cache misses using techniques that include multi-level caches [18], victim caches [17], and prefetching [26]. Reducing the frequency of cache misses also works to reduce the performance impact of cache misses; approaches along these lines include set-associative caches [21], column-associative caches [1], stride tolerant address mappings [10], page coloring [20], and program restructuring to improve data [4] or instruction cache performance [28].

In this paper, a novel software-based data placement optimization, called *Cache-Conscious Data Placement* (CCDP), is introduced as a technique for reducing the frequency of data cache misses. To apply the approach, a program is first profiled to characterize how data is used. The profile information then guides heuristic data placement algorithms in finding a placement solution that decreases predicted inter-object conflict, and increases predicted cache line utilization and block prefetch. The generated placement solution specifies the location of global, stack (local variables), heap, and constants. Placement of global variables and the start of the stack are implemented at compile time using a modified linker. Heap variable placement is implemented at run time using customized allocation routines.

The remainder of this paper details the design, implementation, and analysis of cache-conscious data placement. Section 2 motivates the approach by demonstrating how variable placement can affect data cache performance. Section 3 describes the cache-conscious data placement optimization framework. Section 4 describes the methodology used to gather the results for this paper. Section 5 presents performance results of programs optimized with cache-conscious data placement, and Section 6 describes related work. Finally, Section 7 summarizes the contributions of this work.

2 Motivation

Data placement is the process of assigning (virtual) addresses to data objects. In the context of this work, we term an object as any region of memory that the program views as a single contiguous space. Therefore, each global variable (e.g, a scalar, structure, or an array) is treated as a single object, and each allocated heap segment is treated as a single object. In this paper we will use the term object and variable interchangeably. For cache-conscious data placement, we split the objects into four categories and treat each category differently when performing data placement:

1. Stack - All references to the stack are treated as references to one large contiguous stack object. In this study, we chose not rearrange the order of local variables on the stack. Instead, the stack is profiled and placed as a single object. Since most programs have excellent temporal and spatial locality in stack references, this approach has worked well for the programs we examined.

2. Global - Global variables are located in the global data segment and the addresses for these variables are determined during compilation. The data placement algorithm will provide new starting addresses for each global variable.

3. Heap - All objects that are allocated via dynamic memory management (e.g., malloc, realloc) are labeled as heap objects. Cache-conscious data placement is used to determine a preferred starting location in the data cache for these objects. Then customized allocation routines can be used to allocate these objects at these preferred locations.

4. Constants - All loads that come from inside the *text segment* are treated as loads to constant data. In this study we do not move these constant objects, however, during placement their profiling information is used to determine where to place other objects.

An object is assigned an address when it is created. For global variables, addresses are assigned at compile-time, typically when the program is linked. The start of the stack is also set at link-time or load-time. For stack and heap variables, addresses are assigned at run time, when the dynamic storage is allocated.

The address assigned to a data object affects its location in the data cache. An object's address modulo the data cache block size determines its location within a cache block. For a virtually-indexed cache, an object's address modulo the data cache set size determines the cache set into which the variable will reside. Consequently, data placement can be used as a mechanism to control both the contents of a cache block and location within the cache.

With data placement to control the contents and location of data cache blocks, it becomes possible to influence the performance of the data cache. Consider how changing a variables placement affects a data cache miss from each of the three miss classes [14]:

Conflict Misses: Conflict misses occur when the number of frequently referenced blocks of memory map to the same cache set is greater than the associativity of the cache. Blocks that do not fit into the cache set will displace other blocks each time they are referenced. By placing objects with high temporal locality into different cache blocks, the number of cache conflicts will decrease.

Capacity Misses: Capacity misses result when the working set of the program does not fit in the cache. Referenced cache blocks will displace other blocks because there is simply not enough space in the cache to contain all the frequently accessed blocks. By moving infrequently referenced variables out of cache blocks and replacing them with more frequently referenced variables, cache line utilization can be increased. With better utilization of cache lines, the working set of the cache (in cache blocks) may be decreased, and capacity misses may be eliminated. In addition page utilization will increase.

Compulsory Misses: Compulsory misses occur the first time a variable is referenced. If the variable's cache block has not been previously fetched into the cache, a miss will occur. By grouping variables with high temporal locality into the same cache block so that they do not overlap, cache block prefetches will be used more effectively, and compulsory misses may be eliminated.

3 Data Placement

In this section we describe the CCDP optimization framework. There are three main parts to the optimization framework; (1) the profiler, (2) a data placement optimizer, and (3) run-time support for custom allocation of heap objects.

A program to be optimized is first profiled to gather information characterizing its data usage. Two profiles are generated. In the profiles there is an object data structure for each global variable, each allocated heap object, each constant variable, and one object for the stack. The first profile, *Name*, is a profile listing each unique object encountered during execution along with the object's Id (name), reference count, size, and life-time information. The second profile generated is a *Temporal Relationship Graph* (TRG) between different objects. An edge between two objects in the TRG provides an estimation of the number of cache conflicts that would arise if these two objects were overlapped in the same cache line.

Once the Name and TRG profiles are generated they are fed back into the compiler/linker for data placement optimization. The data placement optimizer reorders the global data segment and determines the new starting location for the data segment and the stack. At this point, if heap optimization is performed, customized allocation routines are generated to guide the placement of heap objects. At run-time these customized malloc routines attempt to allocate data at the preferred locations determined by the data placement algorithm. Essential to an accurate customized malloc is the naming strategy used to identify objects.

3.1 Variable Naming Strategy

The data placement framework requires that profile information collected in one run of the program be used to direct variable placement in another run of the program. To implement this binding, profile and placement tools must assign names to all variables. Design of the variable naming strategy is an important consideration because it has a profound effect on the quality of profile information and the effectiveness of variable placement. There are many strategies to choose from. The one chosen should best meet the following two constraints: (1) variable names should not change between runs with different inputs of a program, and (2) computing variable names should incur minimal run-time overheads.

In the implemented framework, global variables are named using their address. This approach works well to satisfy the above listed constraints. A variable at address X in one run of the program is the same variable at address X in another run, provided the program is not recompiled between runs. In addition, global variable names can be computed at compile time with no run-time cost. A similar naming strategy is used for naming stack variables. Since the the stack is profiled and placed as a single object, the entire stack is assigned a single name.

Generating names for heap variables is a more challenging task. Heap object addresses can change with different inputs to the program, making their address an unsuitable name. The approach implemented in this work is based on the naming scheme of Barrett and Zorn [3]. Heap variables are named when they are created (*e.g.*, at calls to `malloc()`) using the address of the call site to `malloc()` combined (with XOR-folding) with a few return addresses from the stack. Similar heap naming schemes were also employed by Lebeck and Wood [22]. This naming approach does a reasonably good job of satisfying the constraints listed above. Since the addresses of call sites and function returns do not change between runs of a program (provided the program is not recompiled), heap variable names do not change between runs. Computing heap allocation names is very efficient, requiring only a few instructions. This approach does, however, have complications that do not arise with global variables. It is possible for concurrently live heap variables to possess the same name. The placement algorithms recognize this possibility and take this into consideration to try to prevent possibly expensive cache conflicts.

3.2 Representing Temporal Relationships

The Name profile just described provides for each object a unique Name, number of time the object was referenced, size of object, and the life-time of the object. For the objects in this profile we need to create a relationship between the objects in order to determine a placement that minimizes cache conflicts.

In order to determine the ordering for placing objects in the cache, a conflict cost metric is needed. This metric should estimate the number of cache misses that would be caused by placing a group of overlapped objects into the same cache line. To create this metric we use the *Temporal Relationship Graph* (TRG) from previous procedure placement work by Gloy et al. [11]. The TRG contains weighted edges between objects, which represents their degree of temporal locality. A TRG edge is between two objects, and the weight is the estimated number of cache misses that would occur if the two objects mapped to the same cache set (but were in different cache blocks). We term the TRG described in this section the *TRGplace* graph, since it is used to calculate the conflict cost metric when placing the objects into the data cache.

The TRG is created during profiling by keeping a queue `Q` of the most recently accessed data objects. When an object `obj` is referenced via a load or store, the queue `Q` is searched for `obj` starting at the front of `Q`. If the object `obj` is found in `Q`, the conflict weight on the TRG edge (obj,X) is incremented for each object `X` from the front of `Q` to `obj`. The TRG edge (obj,X) weight is incremented because a reference to object `X` occurred between two references to object `obj`, thereby creating a temporal relationship. If `X` and `obj` are mapped to an overlapping location in the cache, this will cause a cache miss for `obj`, assuming a direct mapped cache. After the edge weights have been incremented, the current object `obj` is removed from its location in `Q` and placed at the front of `Q`.

When the size of all the objects in `Q` grows larger than a *queue-threshold*, objects are removed from the end of `Q` until the total size is below the queue-threshold. By limiting the total size of objects in `Q`, temporal relationships will not be recorded for old objects removed from the Q. These older objects have a high likelihood of being displaced from the cache due to capacity constraints on the cache. For this study, we used a queue-threshold of twice the size of the data cache, since our results have shown this to provide most of the important relationships.

The TRGplace graph used for data placement is slightly modified from the above description to keep track of relationships on a smaller granularity than objects. One result from the procedure placement study [11] was that it is hard to place large procedures especially if they are larger than the cache. To effectively place large procedures the temporal information needs to be kept track of on a smaller granularity. We found this same result applies to placing data. Therefore, the TRG maintains edges between *object chunks*, rather than between whole objects. Each object is broken into a set of chunks $size(object)/chunk\_size$. The edges in the TRG now represent the temporal relationship between (object,chunk) pairs. When placing two objects in the cache the conflict cost metric is calculated by examining the TRGplace edges between the (object,chunk) pairs that map to each cache block. For the results presented in this paper we used a chunk size of 256 bytes. This size was large enough to keep the TRG within a manageable size, and small enough to allow large objects to be placed. See [11] for the complete details and tradeoffs for building a TRG.

3.3 Data Placement Algorithm

The cache-conscious data placement algorithm uses the profiled TRG, the size of the objects, and the structure of the target cache (i.e., cache size and block size) to eliminate cache conflicts and increase cache line utilization. Figure 1 shows the overall outline of the algorithm. When placing data objects we use a `CACHE` structure, which stores for each cache block (object `ID`, chunk `NUM`) pairs indicating that the chunk `NUM` of object `ID` is mapped to this location in the cache. We can then easily calculate the conflict cost estimate by looking up the edges between all the `(ID,NUM)` pairs that map to the the same cache block in the TRGplace graph. The size of the `CACHE` structure is equal to the size of the hardware cache for which we are placing the objects.

```
Input:   temporal relationship graph
Output:  placement map
Method:  /* read inputs */
         read_TRG_graph();

         /* PHASE 0: split objects into popular and unpopular sets */
         split_popular_unpopular();

         /* PHASE 1: preprocess the heap objects and assign bin tags */
         preprocess_heap_objects();

         /* PHASE 2: place stack in relation to constant objects */
         place_stack_and_constants();

         /* PHASE 3: make popular objects into compound nodes */
         create_compound_nodes();

         /* PHASE 4: create TRG select edges between compound nodes */
         create_TRG_select_edges();

         /* PHASE 5: place small objects together for cache line reuse */
         cache_line_reuse_for_small_objects();

         /* PHASE 6: place global and heap objects to minimize conflict */
         while (there exists a TRGselect edge) {
            edge = max weighted TRGselect edge;
            merge_compound_nodes(edge→n1,edge→n2)
         }

         /* PHASE 7: place global variables emphasizing cache line reuse */
         choose_final_global_placement();

         /* PHASE 8: finished placing variables, write placement map */
         write_placement_map();
```

Figure 1: Outline of data placement algorithm.

PHASE 0. The first phase of the algorithm partitions the objects into popular and unpopular sets. This partitioning has two benefits, (1) it decreases the execution time of the algorithm by concentrating on only the important relationships, and (2) it identifies infrequently used global objects that can be used to fill in gaps generated during the placement of popular globals. The popularity of an object is the sum of the weights of the TRGplace edges that reference it. Therefore, objects with the most temporal relationships with other objects will possess the highest popularity. The placement algorithm works hard to eliminate cache conflicts for these objects. All objects that account for up to 99% of the total popularity of all objects are considered popular, and the rest are unpopular.

PHASE 1. Heap objects are preprocessed, grouping heap objects which have temporal use and allocation locality together into heap allocation bins. Many of these heap objects will not be marked as popular because they are short lived. Section 3.4 describes this phase in detail.

PHASE 2. In this study we chose to keep the constants located in the text segment fixed. As a result, we build up a `Stack_Const` cache structure in a similar manner described in 3.3.1, first placing all the constant data in the cache using their default virtual addresses. We then determine the best starting block in the cache for the stack based the TRGplace conflict metric between the constant and stack (object,chunk) pairs. Once the starting location for the stack is chosen, the (stack,chunk) pairs are added to the `Stack_Const CACHE`, which is then used in phase 5 of the algorithm for placing the globals and heap objects. The algorithm first picks the best starting location for the stack in terms of the constant objects. These locations are then fixed, and next we will determine the best starting locations for the global and heap objects.

PHASE 3. A compound node structure is created for each popular object. Initially, each compound node contains only one object, this is done in preparation of Phase 5 which groups objects to reduce conflict. A compound node is a set of objects that have been grouped together in the cache during data placement.

PHASE 4. An order needs to be chosen for combining compound nodes. The order in which compound nodes are combined is important because as nodes are combined (merged) this adds constraints to the possible placement of nodes to be merged in the future. This is because once objects are combined together into the same compound node, their relative offsets to one another are fixed. To determine the order in which to process the compound nodes, we create a new graph TRGselect with edges between compound nodes. There are several possible ways to create TRGselect. For this paper, we create TRGselect from TRGplace. Each TRGplace edge is between two `(obj1,chunk1)` and `(obj2,chunk2)` pairs with a weight `W`. For each TRGplace edge we create a compound edge in TRGselect between the compound nodes for `obj1` and `obj2` with a weight of `W`, iff `obj1` and `obj2` are identified as popular objects in phase 1. If this edge already exists we increase that edge's weight by `W`. These compound edges are used to form the TRGselect graph.

PHASE 5. To enable cache block reuse we first make a pass over the global objects that are of size less than the cache block size (32 bytes). Using the TRG_place graph, small objects with high temporal locality are placed together into the same cache block. This will allow cache block reuse, and the objects will benefit from prefetching.

PHASE 6. Once the TRGselect graph is created it is used to determine the order in which to process the compound nodes. The highest weighted TRGselect edge between compound nodes is chosen and the two compound nodes are placed in the cache. After the compound nodes are placed they are combined into a single compound node and their edges are coalesced. This part of the algorithm is described in detail in §3.3.1. The compound nodes are combined until there are no more edges left in TRGselect.

PHASE 7. After phase 5 each popular global and heap object have associated with them a preferred starting offset in the cache. For the popular heap objects this offset is used in the custom malloc to allocate predicted heap names to a memory location that maps to this cache offset. For the global objects this offset is used to determine the ordering for global objects. The global objects are combined in an order to achieve cache block reuse; this algorithm is described in more detail in § 3.3.2.

PHASE 8. Finally, the linking of the global objects in the new order and the custom malloc routine are created, along with a starting location for the stack.

3.3.1 Determining Cache Placement (Phase 6)

Figure 2 shows the algorithm used to combine compound nodes. The algorithm works to eliminate cache conflicts between global, heap, stack and constant objects. As described earlier, the `CACHE` structure contains a set of blocks and each block contains a list of (object,chunk) pairs mapped to that block. The algorithm starts by mapping the two compound nodes `n1` and `n2` to a `CACHE` structure `c1` and `c2`. Once all the objects in each compound node are mapped to the two `CACHE` structures, the algorithm can then go block by block calculating the conflict cost metric using the TRGplace graph. The goal is to determine the starting location for the second compound node `CACHE`, `c2`, in relationship to the first

```
Procedure: merge_compound_nodes(compoundNode n1, compoundNode n2)
Input:     compound node n1 and n2 from TRGselect edge
Output:    a merged compound node and merged TRGselect edges
Method:    CACHE c1, c2;

           if (n1 has never been processed) {
              find location for n1 in relationship to stack and constants;
              adjust offsets in n1 to reflect new starting location;
           }

           foreach obj in n1
              place (obj,chunk) pairs in c1
           foreach obj in n2
              place (obj,chunk) pairs in c2

           preferred_start = choose_intelligent_initial_starting_point();
           best_offset = preferred_start;
           best_cost = infinity;

           for (i=0; i < NUM_CACHE_LINES; i++) {
              start_loc = (preferred_start + i) % NUM_CACHE_LINES;
              cost = 0;
              for (j=0; j < NUM_CACHE_LINES; j++) {
                 fixed_index = (start_loc + j) % NUM_CACHE_LINES;
                 cost += cost_placing_same_block(c1[fixed_index],c2[j]);
                 cost += cost_placing_same_block(Stack_Const[fixed_index],c2[j]);
              }
              if (cost < best_cost) {
                 best_cost = cost;
                 best_offset = start_loc;
              }
           }

           foreach obj in n2 {
              adjust starting cache offset for obj using best_offset;
              merge obj into n1;
           }

           coalesce_outgoing_TRG_select_edges(n1,n2);
           delete n2 from graph;
```

Figure 2: Algorithm for combining compound nodes. The goal is to find the best location in the cache to merge the heap and global object in compound node n2 with the objects in n1 and the `Stack_Const CACHE`.

compound node `CACHE`, `c1`, which has the minimum number of cache conflicts. The cost of each starting location for `c2` is calculated by going line by line through the each of the three caches (`Stack_Const`, `c1`, `c2`) and calculating the estimated number of conflicts for each block using the TRGplace graph.

The first step of the algorithm checks to see if compound node `n1` has been processed. If it has not, then it is first placed with respect the `Stack_Const` cache, in order to eliminate conflicts with the stack and constant objects. Otherwise the compound node `n1` has already been placed, and its offsets have already been adjusted to eliminate stack and constant object misses.

After the main `for` loop in Figure 2 has finished executing, the least cost starting location for `c2` has been found, and the offsets for all the objects in `n2` are adjusted to represent the new placement. Then the two compound nodes are merged into a single compound node, and the TRGselect edges between `n1` and `n2` and the other compound nodes are coalesced. If there is a TRGselect edge (`n1`,`n3`,`x`) and (`n2`,`n3`,`y`), where `x` and `y` are the edge weights, this results in a single edge in the TRGselect graph (`n1`,`n3`,`x`+`y`) assuming `n2` has been merged into `n1`. In this paper we model an 8K direct mapped data cache with 32 byte blocks, and our `CACHE` structure contains 256 lines of size 32 bytes.

3.3.2 Choosing Final Global Ordering (Phase 7)

The final ordering for the global objects is picked to eliminate cache conflicts, group together popular objects, and increase cache line utilization. A final ordering for the global objects, starts by finding the most popular global object and using this to initialize the start of the global data segment. The global objects are then searched for a popular object that has a preferred offset adjacent to the ending offset of the previously processed global. If several candidates exist, the one with the highest temporal locality with the previously placed popular object is chosen. If no popular object can be placed adjacent to the last placed popular object, a gap is created between popular objects. When this occurs, the popular object closest to the end of the previous placed global is chosen to be the next popular object placed. The gap created between the last placed object and the preferred location of this new popular object is filled with unpopular global objects. After all the popular objects have been placed, the unprocessed unpopular objects are placed in the order of most frequently referenced to least frequently referenced.

3.4 Custom Allocation of Heap Objects

Heap allocation placement is implemented at run-time using a customized malloc routine. The modified `malloc` first computes the heap allocation name, an integer value, by XOR-folding `N` return addresses from the stack. For the results in this paper we used a name depth of 4, which other researchers have also found to have reasonable results [30].

The heap allocator we model is similar to previously proposed heap allocators that map objects of similar sizes to the same pages of memory during allocation [12]. The difference is we use data placement to guide heap objects into allocation bins. In the CCDP custom allocator, there are several free lists each with an associated bin tag. When an object is custom allocated and there is a corresponding tag for its XOR name, the object is allocated from the free-list associated with that tag. This strategy allocates objects with temporal locality near each other in memory.

After the TRGplace and Name profile are generated and before any placement occurs, CCDP performs some preprocessing on the heap objects in Phase 1 of Figure 1. Heap objects with temporal use and allocation locality are assigned the same allocation bin tag. Objects with the same tag will use the same free list for allocation and benefit from potentially being allocated close to one another. In addition in Phase 1, all objects that do not have a unique XOR name are marked as unpopular, but they can still benefit from the custom malloc if the object XOR name was assigned a heap allocation bin tag. Therefore, only the popular heap objects with unique XOR names are passed to the placement algorithm to eliminate cache conflicts with the stack and global data objects. Besides possibly being assigned an allocation tag, these popular heap objects are given a preferred cache start offset. When an object is allocated, if an allocation bin tag is found, the heap free list corresponding to that tag is used to allocate the object. If there is no tag, then the default free list is used. During custom allocation, if the object has a preferred cache offset the object is allocated in the free list (chosen by the bin tag or default free list) so that the start of the object maps to the preferred cache block.

4 Methodology

To perform our evaluation, we collected information for 6 of the SPEC95 programs, and 2 C++ programs (`deltablue` and `groff`), and `espresso`. The C and FORTRAN programs were compiled on a DEC Alpha AXP-21164 processor using the DEC C and FORTRAN compilers. The C++ programs were compiled with GCC. We compiled the SPEC benchmark suite under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`).

For the results in this paper we used ATOM [32] to instrument the programs, gather the Name and TRG profiles, perform the data placement optimization, and finally gather the data cache simulation miss rate results. The ATOM instrumentation tool has an interface that allows the elements of the program executable, such as instructions, basic blocks, and procedures, to be queried and manipulated. In particular, ATOM allows an "instrumentation" program to navigate through the basic blocks of a program executable, and collect information about registers used, opcodes, branch conditions, and perform control-flow and data-flow analysis. ATOM allows access to the program's structure but not the data. Therefore, we get the location and size of each global data variable by parsing the symbol table, and reading that into the ATOM instrumentation code.

The Name and TRG profiles produced by ATOM are used by the data placement optimization framework to generate (1) a new ordering for global objects and a new starting address for the data segment, (2) the new starting stack address, and (3) a lookup table of XOR heap Names and predictions for customized malloc. We then simulate the programs to gather their data cache miss rates using this new placement by mapping each old address given by ATOM to the new global, stack, or custom-allocated heap address.

When building profiles and performing the data cache simulations, we track data object allocation and deallocation by instrumenting `malloc`, `free` and `realloc`. We treat each `realloc` as a `malloc` followed by a `free`. During cache simulation, the instrumented `malloc` routine calls our custom malloc, which computes the allocation's XOR name by XOR-folding the 4 most recent call sites; it then uses this value as an index into our customized allocation table. If the XOR name is found, the allocation table would return a customized bin tag and/or a preferred starting location for the heap object. If the bin tag is valid, the object will be allocated from the bin tag free list. If the preferred starting cache offset is valid, the object will be allocated at a free space starting at that offset. If there is no match in the custom allocation table, we allocate the object from the default free list.

5 Results

For the results presented in this paper, we only applied heap placement to `deltablue`, `espresso`, `groff`, and `gcc`. For the remaining 5 programs, we only applied the dataplacement to the constant, stack and global variables. Therefore, these 5 programs have no run-time overhead associated with the use of CCDP dataplacement.

Table 1 shows the two data sets we used in gathering results for each program, along with the program's characteristics. The first input listed for each program is the *training* data set, and the second input listed is the *testing* data set. We created the TRG and Name profiles for the first input and used these to perform the data placement optimization. The new data placement mapping is then used to report the data cache miss rate results for both the first input and second input. Using the same data set for both testing and training provides an ideal performance, and making them different provides realistic performance. In the next section we provide results for both.

The third column in Table 1 shows the number of instructions executed for each input in millions. The fourth column shows the percent of instructions executed that were loads, and the fifth column is the percent of instructions executed that were stores. The next four columns show the percent of all load and store references that where to the Stack, Global, Heap and Constant objects. For example, the results for `gcc` using data set `1recog` has 49% of its executed memory references to the Stack, 21% to the Global data segment, 27% to the Heap, and 2% to Constants in the text segment. The last four columns in Table 1 show the number of executed allocations (calls to malloc), the average allocation size, the number of deallocations (calls to free) and the average deallocation size.

Program	Input	Instr (M)	% Lds	% Sts	% Stack	% Global	% Heap	% Const	Num Malloc	Avg Size	Num Free	Avg Size
deltablue	short	32	28	10	31	19	45	0	37292	38	30817	26
deltablue	long	96	28	10	47	24	20	1	110106	50	91178	28
espresso	short	25	23	6	14	28	56	0	24743	71	24638	70
espresso	cps	513	22	5	7	15	76	0	411757	96	411548	92
groff	me	51	23	11	51	19	23	7	10633	88	8259	75
groff	man	52	23	11	51	20	21	7	12547	79	10678	68
gcc	1recog	264	19	9	49	21	27	2	16844	827	16522	823
gcc	1stmt	337	24	11	51	23	23	2	34244	745	33726	731
compress	cshort	12	22	9	15	85	0	0	1	16	0	0
compress	ref	117	21	7	14	86	0	0	1	16	0	0
go	2stone9	699	22	6	37	63	0	0	4	3084	3	4096
go	5stone21	41678	22	6	35	65	0	0	4	3084	3	4096
m88ksim	train	159	15	9	54	35	11	0	15	531367	7	18081
m88ksim	ref	90508	16	9	46	51	3	0	28	1412988	20	1933823
fpppp	train	311	25	10	65	34	0	1	1175	84	26	1664
fpppp	ref	162866	25	9	68	31	0	1	1175	84	10	2050
mgrid	train	9271	36	3	0	100	0	0	1218	153	69	1905
mgrid	ref	69167	38	2	0	100	0	0	1170	75	5	2050

Table 1: Statistics for data sets used in gathering results for each program.

	Original Placement					CCDP Placement					Percent Reduction
		Miss Rate by Object					Miss Rate by Object				
Program	D-Miss	Stack	Global	Heap	Const	D-Miss	Stack	Global	Heap	Const	
deltablue	21.79	0.38	0.57	20.79	0.05	20.84	0.30	0.47	20.01	0.06	4.36
espresso	3.11	0.49	1.16	1.46	0.00	2.43	0.13	0.49	1.82	0.00	21.64
gcc	8.47	1.10	2.04	4.76	0.56	7.28	0.63	1.33	4.75	0.56	14.06
groff	6.49	1.47	1.51	2.60	0.91	3.63	0.71	0.61	1.55	0.77	44.01
compress	10.92	0.14	10.78	0.00	0.00	7.38	0.12	7.27	0.00	0.00	32.40
go	9.66	1.57	8.09	0.00	0.00	6.26	0.32	5.93	0.00	0.01	35.20
m88ksim	5.20	0.19	3.52	1.49	0.00	1.93	0.13	0.35	1.45	0.00	62.93
fpppp	5.80	1.80	3.70	0.00	0.29	2.41	0.42	1.69	0.00	0.31	58.40
mgrid	7.60	0.01	7.58	0.00	0.01	7.59	0.01	7.57	0.00	0.01	0.13
Average	8.78	0.79	4.33	3.46	0.20	6.64	0.31	2.86	3.29	0.19	30.35

Table 2: Data cache miss rates using the first input to both create the profile and placement and to gather the miss rates. Results are for an 8K direct mapped cache with 32 byte lines.

5.1 Data Cache Performance

Tables 2 and 4 show the data placement performance in terms of the improvement in cache miss rate. Results are shown for the Original program and for cache-conscious data placement (CCDP) for a direct mapped 8K cache with 32 byte lines. The column labeled `D-Miss`, is the percent of overall data cache misses. The four columns following `D-Miss` show the miss rate broken down across our four different types of data (Stack, Global, Heap and Constant). When a cache miss occurs the data object that is referencing memory is assigned blame for that cache miss. The last column shown in these tables is the percent reduction in miss rate achieved for CCDP in comparison to the original placement.

Table 2 shows the miss rates using the first input in Table 1 to both generate the CCDP placement and gather the miss rate results. The results show a large reduction in the data cache miss rate from 8.7% down to 6.6%, a 30% reduction in miss rate. For the original placement, the data objects that cause the largest problem are the global variables. The CCDP algorithm is able to reduce these misses by 34%. For the programs we examined, correctly determining the starting location for the stack has a big impact on performance. The CCDP placement algorithm places global and heap data objects so that the do not interfere with the stack when there is a high degree of temporal locality. This allows CCDP to reduce the number of misses cause by stack references by 61%. The heap provided the smallest improvement of all, with only an 5% reduction in miss rate.

Table 3 breaks down the frequency of references to objects by size in bytes. Each column represents the objects that have a size within the range (in bytes) of the column header. The first column shows the total number of static objects referenced during execution. The first number in each column shows the number of objects of that size. The first number in parenthesis indicates the total percent of dynamic references to objects of that size, and the second number indicates the average percent of all references directed to *each* object of that size. This second number is calculated by taking the percent of dynamic references to objects of that size and dividing this by the number of objects of that size. For example, `compress` has 4 objects referenced between the sizes of 128 bytes and 1024 bytes. These 4 objects account for 22% of the dynamic data references for `compress` . Therefore, each of these 4 objects accounts on average for 5.5% of the data references.

A number of observations regarding the capability of our algorithm can be made by comparing the cache performance impacts from Table 2 with the size statistics of Table 3. The smallest improvement in cache performance was for `mgrid`. This was due to most (100%) of the references being directed to a single object much larger than the cache. As a result, most of the misses are intra-variable misses, and our placement algorithm can do little to help cache performance. For objects of this size other compiler techniques like blocking or tiling could be used to reduce cache misses. These optimizations change the access patterns to the data to eliminate capacity and conflict misses. A large object could even potentially be broken into smaller objects, but this would only be possible if all the code that traverses the object could be identified and also modified. Most of the programs our algorithm performed well on had a set of popular objects with most of them smaller than the cache, *e.g.*, `compress`, `m88ksim`, and `fpppp`. In these cases the algorithm can allocate variables effectively within the limited cache space.

Figure 3 shows the behavior of the heap objects for `deltablue`, `espresso`, `groff`, and `gcc`, and shows the challenge of performing effective CCDP heap placement for these programs. Each point in the graph represents an allocated heap object. The Y-axis shows the percent miss rate for the object, and the X-axis shows the number of times the object was referenced. These results show that most of the objects that have a large miss rate are only reference a handful of times. These objects tend to be small, short-lived, and they have a high miss rate. The accumulated reference count of these objects accounts for most of the heap-based cache misses seen in the simulation results. This is why our current approach to CCDP is not as effective for heap objects.

To obtain a more realistic view of the performance of CCDP, Table 4 shows the miss rates for the second input, using CCDP guided by profiles from the first input. The results show a 24% reduction in the average miss rate.

Although not shown in the tables, we also compared the performance of *random* placement to natural (original) placement. With random placement, we simply map global and heap objects into memory with arbitrary order. Strikingly, we found most programs suffered significantly more data cache misses with random placement, often showing increases of 20% or more. This result clearly shows that natural placement is not a bad one - programs already contain some level of temporal and spatial locality *between* variables. This is likely the result of programmers textually grouping logically related variables, more often than not they end up near each other in memory as well. In light of this result, we were very encouraged that our placement algorithm consistently improved data cache performance, even when using different inputs to test and train.

While we describe our approach in the context of improving data cache performance, other levels of the memory hierarchy can benefit from data placement optimizations as well. Table 5 shows the data cache miss rates (from Table 4) and total virtual memory sizes and working set sizes. The Total column shows the total number of 8 KByte pages used during execution. The working set size is computed using a window (*tau*) of 1% of total execution time. A few programs saw little impact on the working set size, while most of the programs saw the working set size slightly increase. The working set size can actually increase because we are concentrating on eliminating cache misses and not page reuse. Our placement algorithm could be further tuned to improve specifically virtual memory performance, this is an area of future study.

The main reason for an increase in page usage for the heap program is the allocation algorithm. The original placement uses a single heap bin with a first-fit heap allocator [12]. Whereas our heap placement algorithm uses a temporal-fit heap allocation algorithm, and several different allocation heap bins might be used as specified by the placement algorithm. Temporal-fit sorts the free chunks by the last time a chunk of free memory was touched, instead of by size as in a first-fit allocator. For temporal-fit, a free chunk of memory is touched if either of its sides are allocated or if part of the free chunk is deallocated. When we would allocate an object, the first most recently touched chunk of free memory that the object would fit in would be used. We examined several different heap placement allocation algorithms which concentrate on temporal-fit and spatial-fit, but they provided similar cache and page placement results. We are currently investigating different allocation algorithms to help improve cache and page heap placement performance.

5.2 Placement for Multiple Cache Configurations

It is ideal to perform data placement once for a given cache configuration, but an executable can be run on a line of processors with potentially different cache configurations. In the placement phase, variables are sorted to minimize inter-variable conflicts for the target cache geometry. Since variable placement occurs at compile time, the target cache geometry should be selected as the smallest cache size on which the developer expects to attain good (or tolerable) performance for the program being optimized. If the target

Program	Static Num	0 < <= 8	8 < <= 128	128 < <= 1024	1024 < <= 4096	4096 < <= 8192	8192 < <= 32768	> 32768
deltablue	37347	3254 (8, 0)	30843 (40, 0)	3243 (37, 0)	4 (9, 2)	2 (6, 3)	1 (0, 0)	0 (0, 0)
espresso	24617	9461 (13, 0)	12936 (42, 0)	2071 (26, 0)	139 (1, 0)	8 (8, 1)	2 (9, 4)	0 (0, 0)
gcc	17436	511 (8, 0)	10480 (9, 0)	2268 (6, 0)	4080 (55, 0)	71 (3, 0)	24 (3, 0)	2 (14, 7)
groff	10822	147 (12, 0)	9720 (26, 0)	869 (30, 0)	47 (4, 0)	19 (1, 0)	19 (18, 1)	1 (10,10)
compress	51	30 (18, 1)	12 (14, 1)	4 (22, 5)	1 (25,25)	0 (0, 0)	2 (7, 3)	2 (14, 7)
go	315	100 (13, 0)	79 (0, 0)	32 (11, 0)	84 (23, 0)	6 (3, 0)	11 (18, 2)	3 (33,11)
m88ksim	150	80 (21, 0)	15 (1, 0)	26 (28, 1)	12 (2, 0)	2 (0, 0)	8 (22, 3)	7 (27, 4)
fpppp	1224	25 (0, 0)	1170 (4, 0)	12 (0, 0)	4 (84,21)	3 (1, 0)	7 (7, 1)	3 (4, 1)
mgrid	1213	24 (0, 0)	1166 (0, 0)	10 (0, 0)	4 (0, 0)	4 (0, 0)	4 (0, 0)	1 (100,100)

Table 3: Breakdown of the frequency of references to objects in terms of their size in bytes. Each column represents the objects that have a size within the range of the column header. For each column, the first number is the percent of static global and heap objects executed of that size. The first number in parenthesis is the percent of dynamic references accounted for by objects of that size. The next number is the average percent of references per object of that size.

	Original Placement					CCDP Placement					
		Miss Rate by Object					Miss Rate by Object				Percent
Program	D-Miss	Stack	Global	Heap	Const	D-Miss	Stack	Global	Heap	Const	Reduction
deltablue	20.90	0.31	0.47	20.10	0.02	20.45	0.32	0.45	19.63	0.06	2.15
espresso	5.74	0.42	0.52	4.79	0.00	5.41	0.20	0.31	4.90	0.00	5.68
gcc	7.66	1.19	2.26	3.58	0.62	6.28	0.65	1.50	3.52	0.61	18.05
groff	5.90	1.31	1.59	2.00	1.00	4.76	0.91	0.92	1.99	0.94	19.24
compress	15.21	0.21	15.00	0.00	0.00	12.11	0.18	11.93	0.00	0.00	20.41
go	11.46	1.07	10.38	0.00	0.01	10.21	0.52	9.68	0.00	0.01	10.95
m88ksim	3.23	0.18	2.64	0.41	0.00	0.83	0.05	0.40	0.37	0.00	74.41
fpppp	5.84	1.97	3.57	0.00	0.30	2.17	0.39	1.50	0.00	0.28	62.84
mgrid	5.32	0.00	5.31	0.00	0.01	5.32	0.00	5.31	0.00	0.01	0.00
Average	9.03	0.74	4.64	3.43	0.22	7.50	0.36	3.56	3.38	0.21	23.75

Table 4: Data cache miss rates for the second input, using the first input to guide the data placement. Results are for an 8K direct mapped cache with 32 byte lines.

	Original			CCDP Placement		
		Pages Used			Pages Used	
Program	D-Miss	Total	Working Set	D-Miss	Total	Working Set
deltablue	20.90	378	10	20.45	428	13
espresso	5.74	52	9	5.41	60	11
gcc	7.66	588	26	6.28	604	33
groff	5.90	59	14	4.76	71	17
compress	15.21	93	90	12.11	95	90
go	11.46	78	56	10.21	79	60
m88ksim	3.23	4744	546	0.83	4744	545
fpppp	5.84	64	27	2.17	64	27
mgrid	5.32	966	804	5.32	968	804
Average	9.03	780	176	7.50	790	179

Table 5: Paging results showing the total number of 8KByte pages used during execution and the average number of pages in the working set.

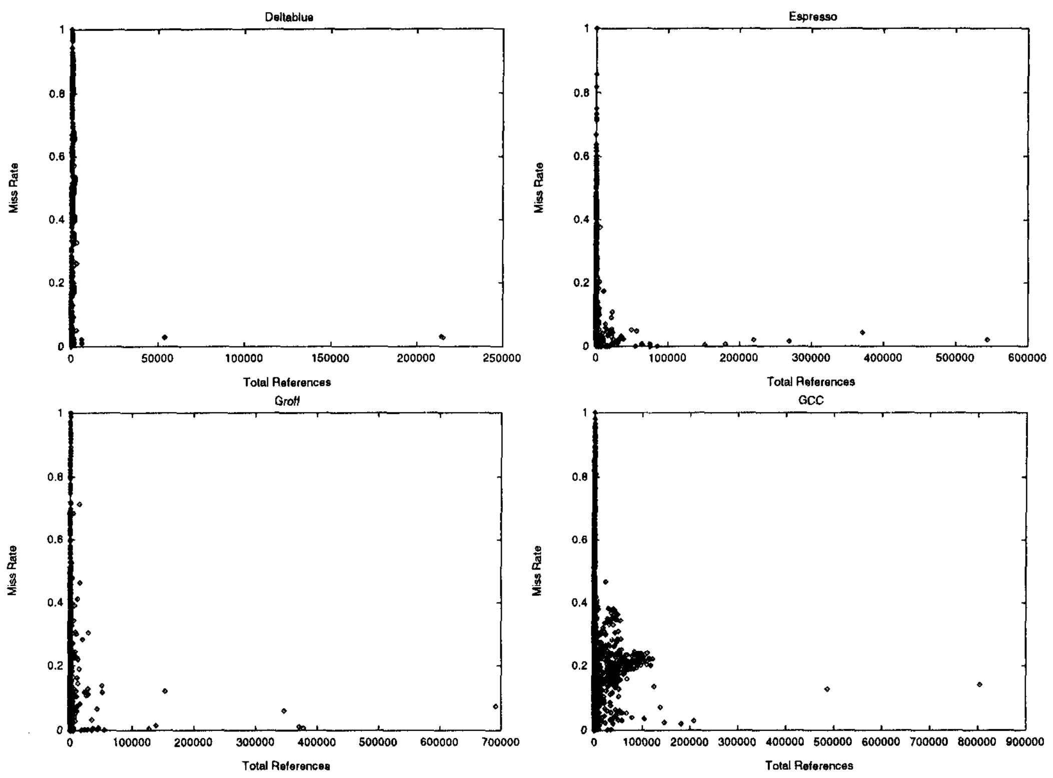

Figure 3: Plot of allocated heap objects, with their cache miss rates on the Y-axis and the number of references to each object on the X-axis. Each point represents an allocated heap object.

cache size selected is too small, the resulting placement will be so strapped by conflict that the placement algorithm will probably not find a good placement solution. If the target cache size selected is too large, the placement algorithm may produce a variable placement solution without considering potentially expensive conflicts in smaller cache sizes.

We are currently extending our placement algorithm to associative caches. The algorithm works the same by placing chunks into cache sets instead of cache lines. The only real change in the algorithm is the cost conflict metric to be used in the TRG graph. An edge in the TRG graph as described in this paper, represents the number of cache conflicts that would occur if the two objects were placed in the same line of a direct mapped cache. Therefore, the TRG graph is representing conflicts for a direct mapped cache. To accurately model an associative TRG, these edges would need to be between sets of objects up to the associativity of the cache. This can be expensive to build during profiling, so we are looking at alternative techniques for gathering this information such as time sampling. Alternatively, the TRG graph for a direct mapped cache may provide enough information to achieve most of the potential from data placement for associative caches.

6 Related Work

A number of peripheral works employ data relocation to improve data cache performance. Page coloring [23, 20] techniques have leveraged the memory mapping capability of virtual memory to reduce conflicts in physically indexed caches. User-programmable cache set mappings [8] have been proposed for similar benefits. Compiler-directed array dimension extension [5] and array variable padding [29] work to relocate data within large arrays, giving opportunity to improve data cache performance when a large array conflicts with itself. Placement optimizations have been used to reduce false sharing in shared memory multiprocessors [16]. Compiler-directed variable partitioning has been proposed as an approach to reduce inter-variable interactions [27] for the purpose of improving the predictability of cache access latencies in real-time systems. The Scout operating system [25] employs data placement to reduce data cache conflict between active protocol stacks.

Many parallels to this work can be found in software techniques developed for improving instruction cache performance. Techniques such as basic block re-ordering [15, 28], function grouping [34, 15, 28], reordering based on control structure [24], and reordering of system code [33] have all been shown to significantly improve instruction cache performance. Like this work, the approaches usually rely on profile information to guide heuristic algorithms in placing instructions to minimize instruction cache conflicts, and maximize cache line utilization and block prefetch.

Recent work on procedure placement for improve instruction cache performance has shown that further improvements in performance can be achieved by keeping track of which cache lines procedures are placed to eliminate conflict misses, and by using temporal information to guide the placement algorithm [13, 11]. This research showed that taking into consideration the cache attributes when performing the placement and the temporal relationships between procedures significantly reduces the cache miss rate. We used the approach presented in [11] for procedure placement as the basis for our cache-conscious data placement algorithm. The data placement mechanisms were adapted from [2].

A report by Seidl and Zorn [30] examined the issues dealing with naming heap allocated objects for potential data placement. Their study examined several different prediction mechanisms used to name heap objects. Their techniques focussed on the XOR naming scheme described earlier in this paper. Their results showed that XORing too many call site addresses can over specialize a custom malloc routine, leading to poor prediction performance between different inputs. They found using an XOR stack depth of 3 to 4 call sites performed well for the programs they examined. They also proposed using the size of the object as an effective means to help distinguish between heap objects that have the same XOR name. Our results confirm theirs, and we use a stack depth of 4 when calculating the XOR heap names. Their study did not quantify the performance of data placement. It was a study focused on techniques to predict heap object names for use in a customized malloc. Our work is different in that our study has focused on developing an overall approach to efficient data placement. Our results show that a holistic approach must be taken to data placement, accurately placing the stack, global, heap, and constants.

Concurrently, Chilimbi *et al.* developed similar techniques for optimizing heap data placement. In [7] they describe a data placement optimization for tree-like structures. Their approach is semi-automatic, permitting more aggressive optimizations, such as the splitting of structure variables. In [6] they extend this approach to support on-line profiling and data placement for Cecil, an object-oriented language with generational garbage collection.

7 Conclusions

Cache-conscious data placement is introduced as a software-based technique to improve data cache performance by relocating variables in the virtual memory space. The approach employs data profiling to characterize variable usage. Profile information then guides compile-time variable placement algorithms in finding a variable placement solution that decreases predicted inter-variable conflicts, and increases predicted cache line utilization and block prefetch. The generated placement solution would then be implemented using a modified linker and customized dynamic allocation routines.

Specifically, our work makes three contributions:

- We present the first general framework for data layout optimization, one which supports analysis and relocation of global, stack, heap, and constant objects. To accommodate this flexibility, we present techniques for naming dynamic objects, profiling temporal relationships, and placing dynamic objects with compile-time placement decisions that are cache-conscious.
- We motivate the need for a highly-capable holistic approach to data layout optimization. We found through simulation that random placement performs significantly worse than natural placement - this sets the bar high for data placement algorithms. Furthermore, we show that data caches misses arise from interactions between all segments of the program address space, necessitating a placement approach that can accommodate relocation of global, stack, and heap objects.
- We couple our layout optimization framework with an effective data placement algorithm. Adapted from previous work on text layout optimization [11], we demonstrate that the algorithm finds placement solutions that improve data cache performance, with a 24% miss rate reduction on average. Moreover, it consistently improves data cache performance across all experiments, even when profiling inputs different from analyzed inputs.

Future work entails implementing CCDP to examine execution performance. Benefiting from heap placement with using XOR names will require a very efficient implementation. For the results we reported, 5 programs (`compress`, `go`, `m88ksim`, `fpppp`, and `mgrid`) did not use heap placement or XOR naming. For these programs, there is no run-time overhead execution cost after CCDP is applied, since the stack and global data objects are placed at compile time. For these programs, their average 26% reduction in cache miss rate should help program performance.

Acknowledgments

The authors would like to thank Guri Sohi for his help in developing the ideas presented in this paper. We would also like to thank Jim Larus and the anonymous referees for their useful comments. This work was funded in part by NSF CAREER grant No. CCR-9733278, UC MICRO grant No. 97-018, and a DEC external research grant No. US-0040-97.

References

[1] A. Agarwal and S. D. Pudar. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 21(2):179–190, May 1993.

[2] Todd M. Austin. *Hardware and Software Mechanisms for Reducing Load Latency*. TR 1311, Computer Sciences Department, UW–Madison, Madison, WI, April 1996.

[3] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, 28(6):187–196, June 1993.

[4] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 28(5):252–262, October 1994.

[5] C.-L. Chen and C.-K. Liao. Analysis of vector access performance on skewed interleaved memory. *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 17(3):387–394, May 1989.

[6] T. Chilimbi and J. Larus. Using generational garbage collection to implement cache-conscious data placement. Submitted for publication., 1998.

[7] T. Chilimbi, J. Larus, and M. Hill. Improving pointer-based codes through cache-conscious data placement. Submitted for publication., 1998.

[8] F. Dahlgren and P. Stenstrom. On reconfigurable on-chip data caches. *Proceedings of the 24th Annual International Symposium on Microarchitecture*, 22(1):189–198, November 1991.

[9] K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 22(2):211–222, April 1994.

[10] Q. S. Gao. The chinese remainder theorem and the prime memory system. *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 21(2):337–340, May 1993.

[11] N. Gloy, T. Blockwell, M.D. Smith, and B. Calder. Procedure placement using temporal ordering information. In *30th International Symposium on Microarchitecture*, December 1997.

[12] D. Grunwald, B. G. Zorn, and R. Henderson. Improving the cache locality of memory allocation. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, 28(6):177–186, June 1993.

[13] A.H. Hashemi, D.R. Kaeli, and B. Calder. Efficient procedure mapping using cache lince coloring. In *Proceedings of the SIGPLANS'97 Conference on Programming Language Design and Implementation*, pages 171–182, June 1997.

[14] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[15] W. W. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. *Proceedings of the16th International Symposium on Computer Architecture*, pages 242–251, June 1989.

[16] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile-time data transformations. *Proceedings of the Symposium on Principals and Practice of Parallel Programming*, July 1995.

[17] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 18(2):364–373, May 1990.

[18] N. P. Jouppi and S. J.E. Wilton. Tradeoffs in two-level on-chip caching. *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 22(2):34–45, April 1994.

[19] D. R. Kerns and S. J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is unknown. *Proceedings of the 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 28(6):278–289, June 1993.

[20] R. E. Kessler. *Analysis of Multi-Megabyte Secondary CPU Cache Memories*. TR 1032, Computer Sciences Department, UW–Madison, Madison, WI, July 1991.

[21] R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill. Inexpensive implementations of set-associativity. *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 17(3):131–139, 1989.

[22] A. R. Lebeck and D. A. Wood. Cache profiling and the spec benchmarks: A case study. *IEEE Computer*, 27(10):15–26, October 1994.

[23] W. L. Lynch, B. K. Bray, and M. J. Flynn. The effect of page allocation on caches. *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 23(1):222–225, December 1992.

[24] S. McFarling. Procedure merging with instruction caches. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 26(6):71–79, June 1991.

[25] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A communications-oriented operating system. Technical Report TR 94–20, Department of Computer Science, University of Arizona, Tucson, AZ, June 1994.

[26] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. *Conference Proceedings of the Fifth International Symposium on Architectural Support for Programming Languages and Operating Systems*, 27(9):62–73, October 1992.

[27] F. Mueller. Compiler support for software-based cache partitioning. *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, 30(11):125–133, June 1995.

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

[29] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the SIGPLAN'98 Conference on Programming Language Design and Implementation*, June 1998.

[30] M. Seidl and B. Zorn. Predicting references to dynamically allocated objects. CU-CS-826-97, University of Colorado at Boulder, January 1997.

[31] J. E. Smith. Decoupled access/execute architectures. *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pages 112–119, April 1982.

[32] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[33] J. Torrellas, C. Xia, and R. Daigle. Optimizing instruction cache performance for operating system intensive workloads. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 360–369, January 1995.

[34] Y. Wu. Ordering functions for improving memory reference locality in a shared memory multiprocessor system. *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 23(1):218–221, December 1992.