Journal of Systems Architecture 56 (2010) 136–149

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Composition-based Cache simulation for structure reorganization

Keoncheol Shin [a], Hwansoo Han [b,*], Kwang-Moo Choe [a]

[a] Department of Computer Science, KAIST, Republic of Korea
[b] Department of Computer Engineering, Sungkyunkwan University, Republic of Korea

ARTICLE INFO

Article history:
Received 1 December 2008
Received in revised form 7 September 2009
Accepted 11 January 2010
Available online 22 January 2010

Keywords:
Field reorganization
Performance prediction
Cache simulation
Compiler optimization

ABSTRACT

Finding the best data layout has been an ultimate goal of memory optimization. Even with data access profile, heuristic algorithms are needed to reorganize data layout for better locality. The best layout could be found by running the given application with all possible data layouts and selecting the best performing layout. This approach, however, can incur too much overhead, particulary when the number of possible layouts are too many. In this paper, we present a composition-based cache simulation for structure reorganization. Instead of running all possible layouts, we simulate only the primary subsets of layouts and compose the cache misses for all layouts by summing up the cache misses of component subsets. Our experiment with the composition-based cache simulation shows that the differences in the cache misses are within 10% of the full cache simulation for 4-way and 8-way set associative caches. In addition to the cache miss estimation, our heuristic algorithm takes account of the extra instruction overhead incurred by structure reorganization. Our experiment with several structure intensive benchmarks shows the 37% reduction in the L1D read misses and the 28% reduction in the L2 read misses. As a result, the execution times are also reduced by 19% on average.

1. Introduction

Memory hierarchy optimizations are key techniques to achieve a good performance on modern architectures. Even on embedded systems, deep memory hierarchies are employed to handle data-intensive applications. For those applications, compiler optimizations have been viable solutions to improve the locality of data accesses. Established techniques to improve locality include code transformations such as loop tiling and loop permutation [1,2]. The effects of these techniques, however, are limited when programs are too complex and memory access patterns are irregular. Recent studies focus more on data reorganization for better cache performance [3–11]. The principal idea in these researches is to place contemporaneously accessed data near one another in memory. By using such reorganizations, we can load closely related data together into as fewer cache lines as possible without cache line conflicts among them. Particularly, they target dynamically allocated structures, since modern applications heavily use structure objects allocated in the heap. They not only reorganize the relative positions of objects, but also change the internal layouts of fields in structure objects.

In this work, we present a new field reorganization technique which adopts all the effective locality enhancing methods. Our technique adopts well known optimization techniques such as aggregating, compacting, and grouping fields from multiple instances of the same structure type [3,5,9,12]. To find the most promising layout we rely on profiling runs with cache simulations. Pure static approaches we explored in [13,14] also achieve relatively good performance in structure reorganization, but the profile-based approach in this work finds the better layouts for target structures but with the increased overheads in profiling and cache simulation. These overheads, however, can be justified for server applications and embedded applications, since those applications often run for a long period time with similar inputs. Optimizing the performance for those types of applications is worth the all costs. Since finding the best performing layout is still an NP-hard problem, even if we know the exact field access sequence [15], we present a performance estimation technique with the composition-based cache simulation. The resulting layouts provide good field reorganizations for cache locality. Our approach compares the performance for all possible layouts but not executing the programs multiple times with all different layouts. Instead, we separately simulate the cache behavior for all the primary subsets of field layouts during the profile run. After profiling, we can efficiently obtain the cache simulation result of any possible layout by combining the simulation results of primary subsets in the layout. In our performance estimation, we also take account of extra computation overhead due to reorganized field accesses [12].

* Corresponding author. Address: Department of Computer Engineering, Sungkyunkwan University, 300 Cheoncheon-dong Jangan-gu, Suwon 440-746, Republic of Korea. Tel.: +82 31 299 4594; fax: +82 31 299 4921.
E-mail address: hhan@skku.edu (H. Han).

1383-7621/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2010.01.002

The remainder of this paper is organized as follows. We first describe the motivation of our method and the basic ideas of our field reorganization. We then provide a detailed description of our field layout selection based on composition-based cache simulation. Next, we experimentally evaluate our method. Finally, we discuss related work and conclude our paper.

2. Motivation

Many studies have shown that reorganizing fields of key structures improves the performance due to the enhanced locality in memory accesses. If a program frequently accesses a couple of particular fields in structures, we can improve cache locality by collocating these fields in memory. Considering the example code in Fig. 1, we find the `search` function in the example traverses a linked list of *Nodes* and returns *data* field in the first *Node* with a matching *key*. We notice that *key* and *next* fields are accessed every iteration whereas *data* fields are accessed just once when the function finds the matching *key*. According to the access frequencies, we can classify *key* and *next* as hot fields, and *data* as a cold field. It is generally beneficial to fetch and store as many hot fields as possible in the same cache line, since the cache performance can be improved by increasing cache locality.

```
struct Node {
      int   key;       // 4 bytes
      char  data[6];   // 6 bytes
      Node  *next;     // 4 bytes
} *T;

char*  search(int key) {
      struct Node *cur = T;
      while (cur != NULL) {
           if (cur->key == key)
                return cur->data;
           cur = cur->next;
      }
      return NotFound;
}
```

Fig. 1. Motivating example: structure-type definition of Node and *search*() function for the node list.

Rabbah and Palem proposed an interesting dynamic data remapping method for cache locality, called *DDRemap* [9]. Their method separates hot and cold fields by using profiled reference counts and colocates the same fields from multiple structure instances in pre-allocated memory pools. If we apply their *DDRemap* to the *Node* structure in Fig. 1, the same fields from multiple structure instances ($O-1, O_2, \ldots, O_n$) are consecutively located as shown in Fig. 2a. In their method, the fields from the same structure instance are located with a constant interval (*MaxFieldSize* × *MaxObjCnt*, which is marked with a bold line in Fig. 2a). *MaxFieldSize* is the size of the largest field among all fields within the structure. *MaxObjCnt* is the maximum number of structure instances that can be allocated within a memory pool. We can statically calculate *MaxObjCnt*, since the size of a memory pool and the sizes of all fields are known at compile time. This method uses padding spaces between fields to make all field offsets constants and access every field with one load instruction. This layout improves the locality for most cases, but potentially hurt the performance when large padding spaces are added for structures with various field sizes. In addition to their scheme, many researchers found that grouping related fields within a structure also improves the performance further [3,5,10]. When

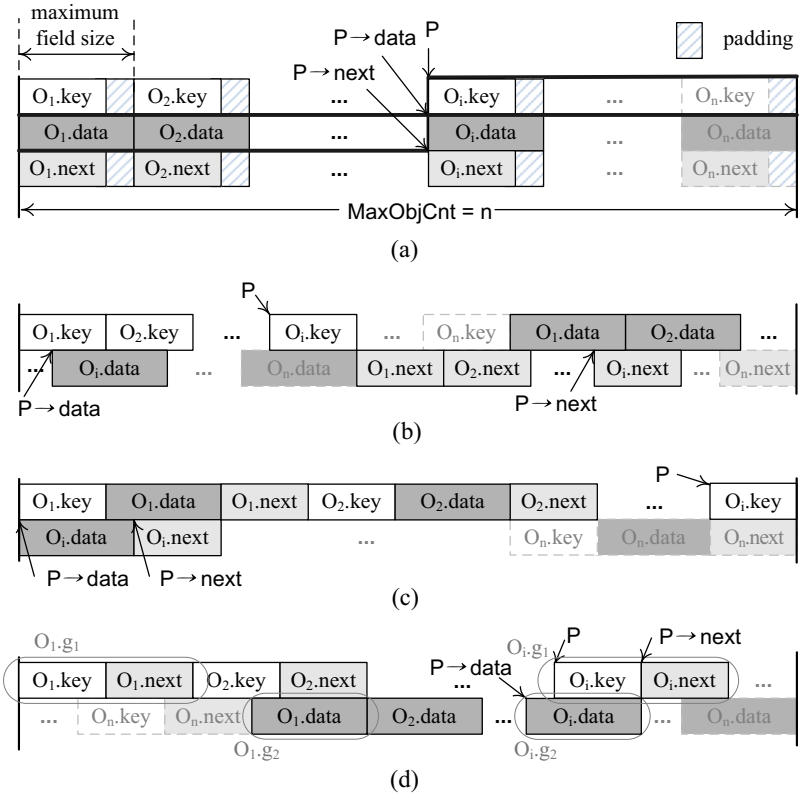


Fig. 2. (a) Structure layout of *DDRemap* [9], (b) structure layout after compacting fields [12], (c) structure layout of pool allocation [16], (d) structure layout after grouping and compacting fields [13,14,12].

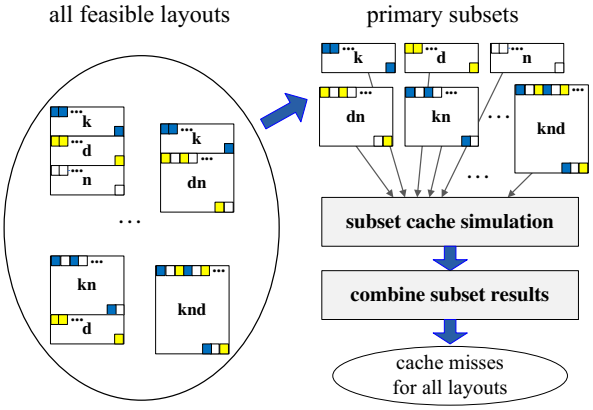


Fig. 3. Composition-based cache simulation: simulate cache with primary subsets and combine the subset results to produce all cache misses.

field grouping is additionally employed to *DDRemap*, making all the groups have the same size becomes more difficult or imposes restrictions on grouping. If field grouping is still intensively used, quite large padding spaces could be inserted.

On the other hand, compacting field schemes, as shown in Fig. 2b and d, eliminate all the padding spaces and still allow compact colocating and grouping without restrictions. These layouts with compacted fields, however, intrinsically require extra instructions for field offset calculations at run time [12]. Nevertheless, they can improve the performance by minimizing the calculation overheads. Pool allocation technique can have a similar field layout shown in Fig. 2c when they use type-homogeneous pools [16]. This layout also compactly uses a memory pool without padding spaces. Fig. 2d is extended from Fig. 2b by combining field compacting with field grouping [13,14,12]. Since the closely accessed fields (*key* and *next*) are placed consecutively in a group, fewer cache lines are required and the resulting layout shows an even further improved locality. All these techniques use custom memory allocations. Large chunks of heap memory, called *pools*, are allocated in advance and individually managed with inexpensive allocation routines. The fields bounded by dotted lines and filled with vague colors in Fig. 2 represent the memory spaces to be used for future allocations. The memory pool scheme reduces management overheads and improves spatial locality by reducing the size of working set.

When we consider field reordering and grouping, finding the most promising layout is very difficult even if we know the access patterns of field from profiling runs. A straight forward approach is running the given program multiple times with all feasible layouts. Although this approach is actually doable for the structures with a small number of fields, this is impractical in general. There can be many target structures and the number of feasible layouts exponentially increases as the number of fields increases. In such cases, our composition-based cache simulation can be used to estimate the performance of all feasible layouts. Fig. 3 shows a rough idea of composition-based cache simulation. First, we extract the primary subsets of the fields that can constitute the all feasible layouts. Next, we simulate the cache only for the primary subsets and combine the simulation results to obtain the cache behaviors of all feasible layouts. In the next section, we will describe how we use this cache simulation scheme to estimate the performance of each layout and select the best performing layout.

3. Performance estimation

The overall process of our structure reorganization is outlined in Fig. 4. The entire process consists of three steps. In the first step, we profile the given program with a small training input and select frequently accessed structures as target structures for field reorganization. We also obtain the sequences of field accesses for the selected target structures. Based on the profiled field access pattern, the second step determines the most promising field layout in terms of cache locality and instruction overhead. It estimates the performance of all feasible layouts and finds the most promising one. In the final step, the code transformer extended from the CIL compiler [17] generates a new source code which utilizes the field layout with enhanced cache locality. In our early works [12–14], we relied on the field affinity graph proposed in [5,6], which approximates the cache behavior by increasing the weight of the edge between two fields when those two fields appear within a predefined distance on the access sequence. Meanwhile, we present a more accurate method to directly capture the cache behavior by using the composition-based cache simulation.

3.1. Composition-based cache simulation

Executing a program multiple times with all possible layouts is often a prohibitively expensive approach, as the number of all

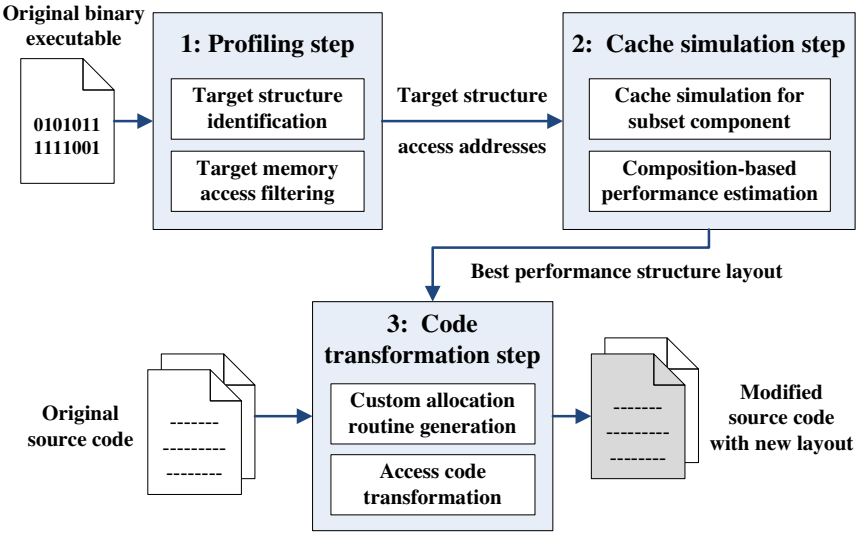
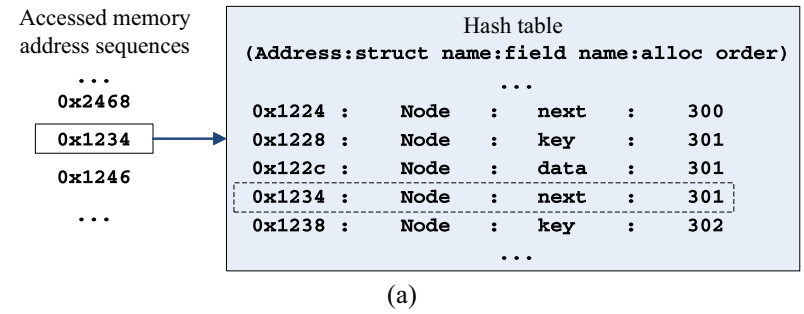


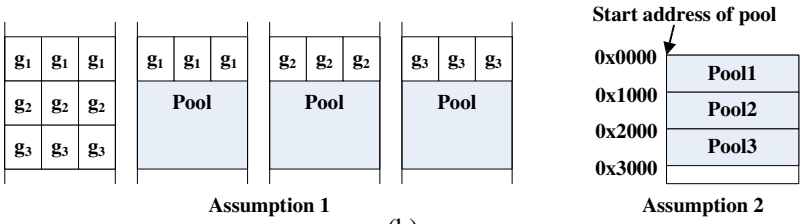
Fig. 4. Overall process of our field reorganization technique.

possible layouts could be too large. Instead of exhaustively running programs, we can compare the performance by simulating only the important components that would vary from one run to another. Since the cache performance is the main component of the performance which would vary depending on field layouts in our case, we simulate the cache to compare the performance among all possible layouts. To expedite the cache simulation, we simulate the cache only with the field accesses from the target structures not the all data accesses. Since the target structures are frequently accessed data, they tend to dictate the performance of the cache. In the experimental section, we will show that other data accesses little interrupt the cache behaviors of the target structures and the cache behaviors of other data also change little across the multiple changed layouts of target structures. Thus, simulating only the target structures is a relatively reliable metric for the cache performance.

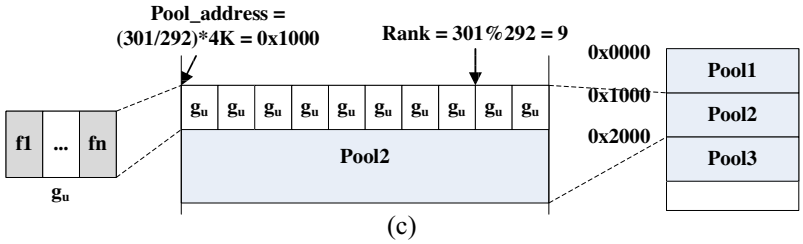
Considering that a field grouping is a combination of the disjoint subsets of fields whose union is the whole structure, we can obtain the simulated cache performance of a given field layout by combining all the separate cache simulations of the field subsets. We align pools to the addresses which are multiples of the pool size. We also make the size of a pool equal to the size of all the cache lines of the same way in each set. For instance, the size of the all the cache lines of the same way for the 32 kB 8-way set associative cache is 4 kB (32 kB/8-ways). Under such configuration, we can guarantee that field groups in a pool do not cause conflict misses with each other. When a program is running, it will access the data not only from one pool but also from multiple pools and also from outside the pools. Since most of modern processors are equipped with set associative caches, such accesses will be accommodated in the cache without causing too much conflicts. With these observations, we first separately simulate distinctive field



(a)



(b)



(c)

All subsets	Virtual addresses of subset for cache simulation
{key}	Null
{data}	Null
{key, data}	Null
{next}	(301 / 292) * 4K + (301 % 292) * 4 = 0x1024
{key, next}	(301 / 292) * 4K + (301 % 292) * 8 + 4 = 0x104c
{data, next}	(301 / 292) * 4K + (301 % 292) * 10 + 6 = 0x1060
{key, data, next}	(301 / 292) * 4K + (301 % 292) * 14 + 10 = 0x1024

(d)

Fig. 5. Virtual address generation for multiple subset cache simulation: (a) hash table lookup for field name and allocation order matching, (b) two assumption for the field address generation, (c) pool address and Rank calculation, and (d) field address generation for cache simulations of all subsets.

subsets from all possible layouts. For the cache simulation result of a particular layout, we combine the separate simulation results of the field subsets that constitute the whole layout we want to simulate. In this manner, we can greatly reduce the simulation times to obtain the cache performance of all possible layouts.

For example, if a program has a structure with n fields, we need to simulate as many times as the total number of all possible layouts to compare the cache performance in a naive approach. Our subset simulation method only simulates $2^n - 1$ times for all non-empty subsets, which is far smaller than the total number of possible layouts when the number of fields is large. All the cache simulation results are then generated by combining the simulation results of the subsets. As shown in Fig. 5d, we only need to simulate seven subsets for a *Node* structure with three fields ($key, data, next$) and combine the simulation results of the subsets to make 10 simulation results for all possible grouping layouts as described later in Table 1c. According to our experiment, subset simulation method takes a reasonable amount of time to finish the simulation of all subsets, which is a far smaller time than executing all possible layouts.

3.2. Cache simulation of all subsets

In the profiling step, we collect all the memory accesses, but filter out many unnecessary memory accesses, since our cache simulator needs only the addresses of the accessed fields in target structures. For the purpose of filtering, we build a hash table during the profile run. We augment the given program at all the allocation sites of target structures and record in the hash table the allocation order of the instance among the same structure type and the allocated addresses for individual fields. Fig. 5a shows how we use the hash table for filtering and cache simulation. For each address from the sequence of target structure accesses we obtained in the profile run, we look up the hash table to find the corresponding field and the allocation order. If there are no entry, the access is not made to target structures and it is unnecessary to our subset cache simulation.

We need to know the accessed field and the allocation order of the structure instance, since these two pieces of information enable us to calculate the virtual address for the subset simulation. For the subset simulation, we concurrently simulate multiple subsets. Thus, we generate multiple virtual addresses for the subsets that contain the corresponding field. When we generate the virtual addresses for our cache simulations, we assume two conditions shown in Fig. 5b, without loss of generality. First, we assume each subset is the first group within its structure (*Assumption* 1). Since we separately simulate each group and no conflict misses occur among different groups in a pool, the placement of the group within a pool does not change the cache behavior of hits and misses. Second, we assume that all the pools for the same structure type are consecutively placed from the address zero in the virtual address space (*Assumption* 2). For the purpose of cache simulation, we do not need the real addresses. We rather need relative distances among memory accesses. In addition to that, the size of a pool is the same as the size of all cache lines of the same way in each set. From the view of cache management, the mappings of all the pools are completely overlapped one another. With this observation, we can safely assume all the pools are consecutively placed from the address zero.

Table 1
Example of performance estimation: (a) cache behaviors obtained from profiling, (b) machine dependent parameters for overhead calculation, and (c) estimation of performance variations for all groupings.

(a) Cache behaviors of all subsets

Group index	Access freq.	L1D hits	L1D misses	L2 misses
1: {key}	1000	950	50	5
2: {data}	100	90	10	1
3: {key, data}	1100	970	130	15
4: {next}	1050	990	60	6
5: {key, next}	2050	1950	100	10
6: {data, next}	1150	1000	150	18
7: {key, data, next}	2150	1960	190	25

(b) Machine dependent overhead

Instructions	Cycles
L1D miss penalty	17
L2 miss penalty	165
AND, ADD, SUB, MOV	0.5
LEA	3
SAL, SAR	4
IMUL	14
IDIV	56

(c) Estimated performance overhead of all grouping combinations

Structure layout		Memory access overhead in cycles (A)	Extra instruction overhead in cycles (B)	Overall overhead ($A + B$)
Subsets	Group order			
7: {key, data, next}	7	17*190 + 165*25	0	7355
3: {key, data},	3,4	17*(130 + 60)+	(14*1 + 4*3 + 0.5*16)*1050	42,395
4: {next}	4,3	165*(15 + 6)	(4*2 + 0.5*6)* 1100	18,795
5: {key, next},	5,2	17*(100 + 10)+	(4*1 + 0.5*2)*100	4185
2: {data}	2,5	165*(10 + 1)	(14*1 + 4*2 + 0.5*16)*2050	65,185
1: {key},	1,6	17*(50 + 150)+	(4*2 + 0.5*6)*1150	19,845
6: {data, next}	6,1	165*(5 + 18)	(14*1 + 4*3 + 0.5*16)*1000	41,195
1: {key}, 2: {data}, 4: {next}	1,2,4		(4*1 + 0.5*2)* 100	4520
	2,1,4	17*(50 + 10 + 60) + 165*(5 + 1 + 6)	(14*1 + 4*2 + 0.5*16)*1000+(14*1 + 4*2 + 0. 5*16) * 1050	65,520
	4,1,2		(4*1 + 0.5*2)*100	4520

From these two assumptions, the field address in a group can be calculated by the following equation.

$$\begin{aligned} &pool\_address(g_u) = \lfloor alloc\_order/MaxObjCnt \rfloor \times Size(pool) \\ &rank = (alloc\_order \ \% \ MaxObjCnt) \\ &address(f_i \ in \ g_u) = pool\_address(g_u) + rank \times Size(g_u) \\ &\qquad + \sum_{k=1}^{i-1} Size(f_k \ in \ g_u) \end{aligned} \tag{1}$$

The $pool\_address(g_u)$ indicates the beginning address of the pool to which the group g_u belongs. In Eq. (1), *alloc_order* represents the allocation order of the structure instance among the instances of the same structure type. *MaxObjCnt* is the maximum number of structure instances we can allocate within a pool. For example, since the sizes of pool and *Node* structure in Fig. 1 are 4096 and 14 bytes, respectively, *MaxObjCnt* is calculated as 4096/14 = 292. The *rank* means the order within a pool. All the variables except for *alloc_order* in this equation are all known at compile time. From the hash table in Fig. 5, *alloc_order* can be determined.

For example, the hash table lookup of the accessed address, 0×1234, returns the structure type and field, *Node.next* and the allocation order, 301 as shown in Fig. 5a . If we apply these numbers to Eq. (1), we can calculate the *pool_address* as $\lfloor 301/292 \rfloor \times 4\,K = 0 \times 1000$ and the *rank* as $(301\%292) = 9$ as shown in Fig. 5c. The third equation consists of the beginning address of the pool (*pool_address*), the group offset within this pool ($rank \times Size(g_u)$), and the sum of the sizes of preceding fields within this group. Fig. 5d shows how we generate the multiple virtual addresses for the primary subsets. Using Eq. (1), our cache simulator generates the field addresses of four different subsets which contain the field, *next* and proceeds to simulate the cache behaviors for those four subsets. For the other three subsets, we do not generate the field addresses, as they do not contain the accessed field, *next*.

3.3. Performance estimation of all subsets

In order to select the most promising field layout, estimating cache misses is not enough. Since some layouts require extra instructions to access fields, we need to take account of those run time overheads as well. In this section, we discuss how to estimate the instruction overheads involved in the field accesses. First thing we need to do for the performance estimation is to generate all possible groupings. This is equivalent to finding all partitions of a set, which means there is no ordering among groups. Even though we do not need the total ordering among groups, we should select the first group in a pool. Since the extra instructions in the calculations of the field offsets are unnecessary for the fields in the first group, the overhead of extra instructions is only applied to the rest of the groups [12].

For example, the first column of Table 1c shows all possible groupings and corresponding group indexes for *Node* structure from our motivating example in Fig. 1. The second column of Table 1c shows the order among groups. The first group has no overhead in field accesses, while the other groups require extra instructions to access fields [12]. Thus, we differentiate only the first group from the rest of the groups and the group order reflects this fact. To calculate the total overheads of a layout, we use the sum of the cache miss penalties (the third column of Table 1c) and the extra instruction cycles (the fourth column of Table 1c). The sum of the two overheads is shown in the fifth column of Table 1c. The cache miss penalties are calculated by using the results from the cache simulation of all subsets, which are shown in Table 1a. The extra instruction overheads in the offset calculations are calculated by using the machine specific parameters obtained from the architecture specification of Intel processors [18], which are shown in Table 1b.

By placing the most frequently accessed group at the first position, we can avoid much of the overhead involved in the field offset calculation [12]. The size of the first group also affects the type of extra instructions in the field offset calculations. If the size of the first group is a power of two, offset calculations can be done with shift operations instead of integer divisions. The fourth column of Table 1c actually shows how to calculate the extra instruction overhead for each subset. For example, the second structure layout in Table 1c consists of two subsets, {key,data} and {next}, and the subset {key,data} comes first. To calculate the memory overhead, we find the number of cache misses for each subset from Table 1a. The numbers of L1D cache misses are 130 and 60 for the two subsets ({key,data} and {next}), respectively. Referring to Table 1b, we find L1D miss penalty is 17 cycles. Thus, the total L1D miss penalty is $17 \times (130 + 60)$. We collect the number of misses for L2 cache and look up the L2 miss penalty in a similar way. The numbers of L2 cache misses are 15 and 6 for the two subsets, respectively and the L2 miss penalty is 165 cycles. The calculation for the total L2 miss penalty is $165 \times (15 + 6)$. Finally, we add up the L1D miss penalty and L2 miss penalty to calculate the memory access overhead as in the third column of Table 1c.

In calculation of the extra instruction overhead, we take account of only the second subset, {next}, as accessing the first subset involves no extra instructions. To access the field *next*, we need to use one integer multiply (IMUL), three shifts (SAL, SAR), and sixteen arithmetic instructions (AND, ADD, SUB, MOV). According to Table 1a, the subset {next} is accessed 1050 times. To calculate the extra instruction overhead, we obtain the average latency of each instruction type from the Table 1b. The latencies are 14, 4, and 0.5 cycles for integer multiply, shift, and arithmetic instruction,respectively. Using the latencies and the numbers of extra instructions, we can calculate the extra cycles to access the field *next* as $14 \times 1 + 4 \times 3 + 0.5 \times 16$. By multiplying the access frequency of the field (1050 for *next*), we can estimate the extra instruction overhead as in the forth column of Table 1c. The following Eq. (2) is the generalized version to calculate the overhead for a given subset. To calculate the total overhead of a layout, we need to add up the overheads of all the subsets, which belong to this layout, as shown in Eq. (3).

$$\begin{aligned} Overhead_k &= memory\_access\_penaltyforsubset_k \\ &\quad + extra\_instruction\_cyclesforsubset_k \\ &= L1D\_misses(subset_k) \times L1D\_miss\_penalty \\ &\quad + L2\_misses(subset_k) \times L2\_miss\_penalty \\ &\quad + \left(\sum_{i \in extra\_instrs}^{\#extra\_instrs} latency_i \right) \times access\_freq(subset_k) \end{aligned} \tag{2}$$

$$Overhead_{total} = \sum_{k}^{subsets\_in\_layout} Overhead_k \tag{3}$$

4. Experimental evaluation

We experimentally evaluate our reorganization technique on a Linux PC with a 1.86 GHz Core2Duo processor and 3 GB main memory. The processor has a 32 kB L1D cache per core and a 2 MB unified L2 cache. Both L1D and L2 caches are configured to use 64 byte cache lines and 8-way set associative mapping. All the benchmarks in our experiment are compiled with gcc 4.1.1 with -O3 optimization level. All the reported execution times are average elapsed times out of five runs. For the cache simulation,

Table 2
Characteristics of benchmark programs.

Benchmarks	Target structures		Inputs		Cache	Code
	#Target structures	#Fields in each	Small (for profile)	Large structures	Cache misses in target structures (%)	Code size increase (binary) (%)
em3d	1	7	6×10^5	7×10^6	62	3.5
health	4	3,4,7,6	10, 20	11, 50	78	8.1
mst	2	3,3	3000	9000	88	4.0
treeadd	1	3	24	27	35	2.5
tsp	1	7	2×10^5	4×10^6	85	3.6
voronoi	1	5	5×10^5	3×10^6	29	1.8
ft	2	5,4	$10^3, 10^5$	$10^3, 4.5 \times 10^5$	96	1.5
181.mcf	2	15,8	Train	Reference	11	0.9

we used the cachegrind from the tool suite of Valgrind version 3.1.0 [19].

To automate the code transformation, we extended the CIL compiler [17]. Once we determine the target structures and their field layouts, our code transformer automatically transforms the source code to use custom (de)allocation routines for target structures. In custom allocation routine, we first allocate a large memory chunk called *bank* and allocate multiple pools within the bank. We vary the size of bank from 400 kB to 4 MB according to the amount of data usage in applications. Our compiler also transforms the field access code to an appropriate code sequence for the new field layout. If fields are accessed through pointer arithmetic operations, we cannot recognize the structure accesses even with a complex alias analysis. Thus, we should check all target structures whether their fields are accessed only by their field names. If any structures violate this rule, they are excluded from the target structures.

We used eight benchmarks from several benchmark suites. *Em3D* uses a linked list for an electromagnetic wave propagation in a 3D object. *Health* simulates the Columbian health care, which heavily uses double-linked lists. *Mst* uses arrays of single-linked lists for a minimum spanning tree of a graph. *Treeadd* performs recursive sum of values in a balanced B-tree. *Tsp* is a famous traveling salesman problem solver, which uses a balanced binary-tree. *Voronoi* computes the voronoi diagram of a set of points recursively on the tree. The six benchmarks mentioned so far come from the Olden benchmark suite [20]. *Ft* is also included due to its pointer-intensive characteristics [21]. 181.*mcf* is a minimum cost network flow solver from the SPEC CPU2000 benchmark suite [22].

Table 2 shows the characteristics of benchmark programs. The second and third columns of Table 2 show the number of target structures selected for our field reorganization and the number of fields within those target structures. The next two columns are the input parameters for the benchmarks used in our experiments. The small inputs are used in profiling and the large inputs are used to evaluate the effectiveness of our profile-based field reorganization. The sixth column shows the percentage of cache misses from the target structures among all the cache misses. This is measured with small inputs. A relatively large portion of the cache misses are originated from the target structures, since our benchmark programs intensively use dynamically allocated data structures. 181.mcf shows only the 11% cache misses from the target structures, but its poor cache behavior notoriously hurts the performance of this program. The last column shows the percentage of the code size increase in binary executables after transforming the programs to use the selected field layouts. Due to the extra instructions for offset calculations and the custom memory management routines, our transformed code increases in the binary size by up to 8% for *health*, but less than 4% for the rest of the benchmarks.

The size increase in *health* is far bigger than the others. *Health* has four target structures and each target structure has its own custom memory (de)allocation routines. These routines increase code size by about 5%. If we provide a parameterized custom memory routine to handle the all four target structures, we could reduce the size increase. In our automatic translation, this is handled rather poorly, but it can be done in a careful translation. We also observed that the selected layouts of two target structures need extra instructions for field accesses at many statements of source code. Those field accesses are, however, not executed frequently at run time, resulting in little increase in dynamic instruction counts. We discuss the changes of dynamic instruction counts later in Section 4.3.

4.1. Accuracy of composition-based cache simulation

In order to justify our cache simulation approach, we compared the cache misses from our composition-based simulation with the ones from the original cache simulation. The number of cache misses in our approach is obtained by summing up the cache misses of the corresponding subsets in that layout. Since we focus on only the cache misses of the target structures, we modified the cachegrind to measure the cache misses only from the target structures. Our cache simulation could be imprecise, if the cache misses from non-target data and other target structures interfere too much. Thus, we devise two metrics, accuracy and disturbance.

For several field reorganization techniques, we obtained the numbers of cache misses for the target structures using two cache simulation methods. We performed our comparison on three different cache configurations in their set associativity. Fig. 6 shows the accuracy of our composition-based cache simulation by displaying the ratios of the L1D misses from two cache simulation methods. The following Eq. (4) defines the accuracy metric with the numbers of misses from two simulation approaches.

$$Accuracy = \frac{\#misses(composition\_based\_simulation)}{\#misses(original\_simulation)} \quad (4)$$

Cmalloc uses a custom pool allocation similar to [16]. An example of this layout is shown in Fig. 2c. *DDRemap* colocates same fields from multiple structure instances as shown in Fig. 2a [9]. *Static* is a static field affinity graph approach used in [13,14] and *Field_Affi* is a profile-based field affinity graph approach used in [5,12] *Perf_Affi* denotes our field reorganization approach, which takes account of the estimated cache misses and the extra instruction overheads. In a direct-mapped cache, our cache simulation method produces inaccurate number of cache misses for each layout. On the other hand, estimated misses in 4-way and 8-way set associative caches result in fairly accurate numbers. Error margins are less than 10% for all layouts of all benchmarks except the *DDRemap* layout of *voronoi*, which is nearly 20%. Inaccuracy in directed mapped cache is expected, since target structures will conflict more with non-target data and even with one another. As our composition

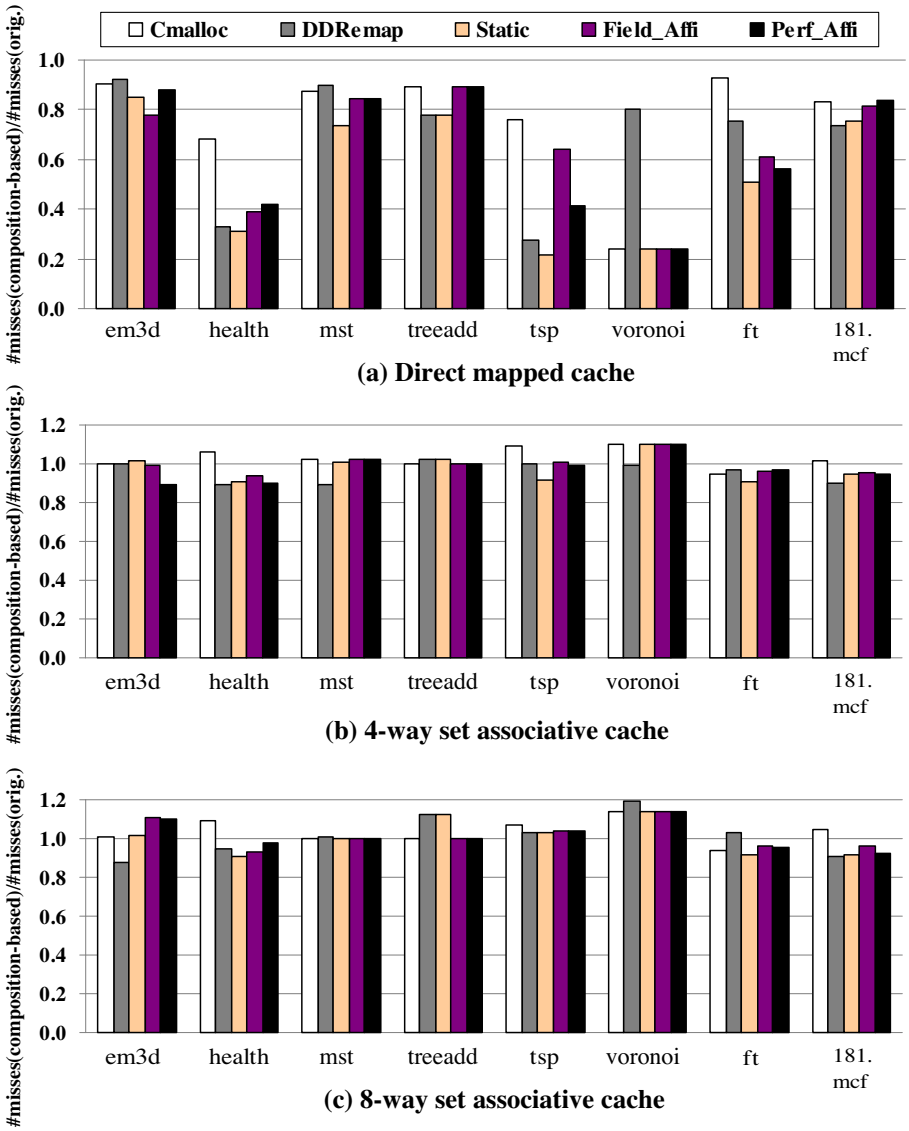


Fig. 6. Accuracy of cache miss in L1D cache simulation for (a) Direct-mapped, (b) 4-way set associativity, and (c) 8-way set associativity.

Table 3
Disturbance of non-target data to cache performance estimation.

Benchmarks	em3d (%)	Health (%)	mst (%)	treeadd (%)	tsp (%)	voronoi (%)	ft (%)	181.mcf (%)
DDRemap	−0.4	−3.5	−1.1	0.7	−3.2	0.1	−2.4	−2.3
Field_Affi	−0.4	−4.2	−2.5	0.4	−4.4	0.4	−2.9	−3.2
Perf_Affi	−0.3	−1.2	−0.3	0.0	−2.8	0.0	−3.1	−1.6

based simulation assumes no conflict misses other than the conflicts in the same subset, our cache simulation method is fitted for set associative caches, which can tolerate conflict misses fairly well. Most modern processors adopt the set associativity in their caches to accommodate conflicting accesses. Thus, we believe our composition-based cache simulation is a quite reliable method to find the most beneficial layout on multi-way set associative caches.

Assuming the cache behavior of non-target data is rarely affected by the layouts of target structures, our performance estimation method only compares the cache performance of target structures. If the cache behavior of non-target data abruptly changes across different layouts for target structures, our cache performance comparison will be incorrect due to the disturbance from non-target data. In order to verify our performance estimation method is valid, we introduce a metric, called disturbance, which shows how much the misses from non-target data change in a new structure layout. Our metric is defined as follows:

$$Disturbance(p) = \frac{\#misses(non\_target, p) - \#misses(non\_target, Cmalloc)}{\#misses(all\_data, p)} \times 100 \quad (5)$$

where $\#misses(D,P)$ is the number of cache misses from the data D, when the layout P is used for the target structures.

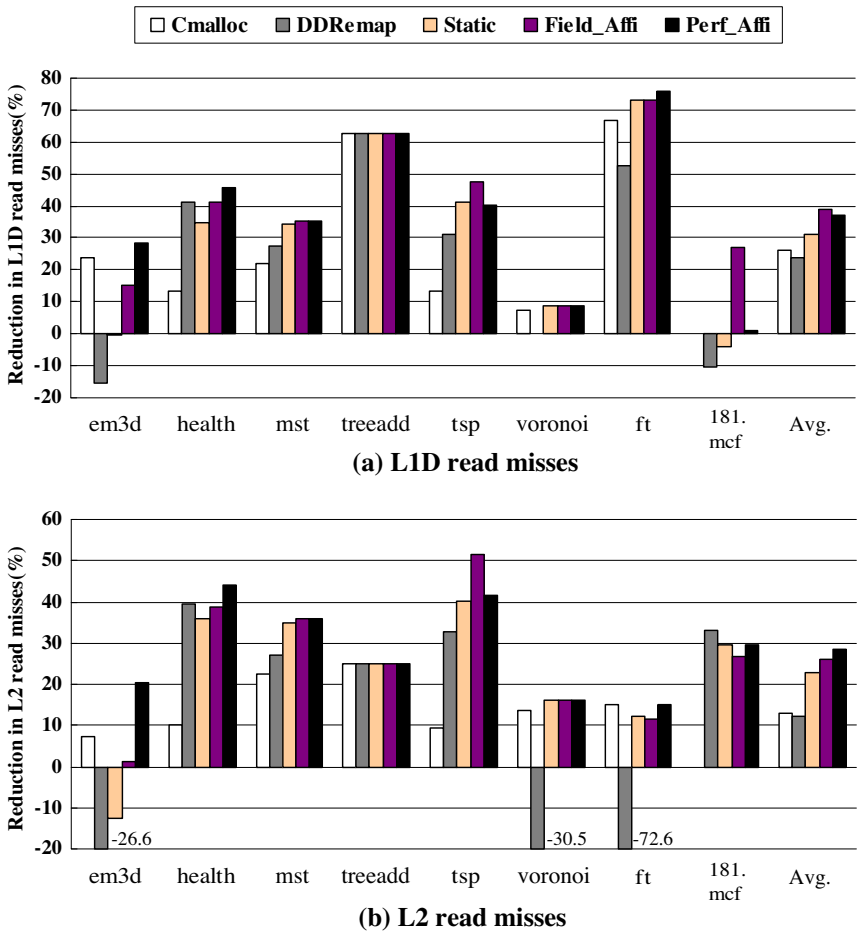


Fig. 7. Reductions in read misses of L1D and L2 cache with large inputs.

The disturbance is the percentage of the cache miss difference in non-target data over all the cache misses from all data. Since *Cmalloc* has the same field layout as the original layout and the pool allocation is used by all candidate layouts, we choose *Cmalloc* as the base layout for disturbance metric. Table 3 shows the disturbances of three different layouts for all benchmarks. The differences in cache misses from non-target data ranges from −4.4% to 0.7% of the cache misses for all data, which means the number of the cache misses from non-target data does not change much from the base case. A negative disturbance means the cache misses from non-target data are actually reduced, when different layouts are used for target structures. Even though our cache miss estimation, which considers only target structure, could be erroneous due to the disturbance of non-target data and the error from composition-based cache simulation for target structures, the sum of two errors was still less than 10% of the actual misses for each tested layout. Only exception is the *DDRemap* layout of *voronoi*, which is 20%.

4.2. Impact on cache performance

In order to evaluate the cache performance of our reorganization technique, we measured the numbers of cache misses for all benchmarks. We simulated the cache behavior with the same configuration as the real machine on which we measured execution times. Fig. 7 shows the reduction in L1D and L2 cache read misses compared to the original layout. The simulation is done with large inputs. CMALLOC reduces read misses of L1D and L2 caches by 26% and 13%, respectively. DDREMAP shows more cache misses than CMALLOC in L1D cache misses for *em3d* and 181.*mcf*, and in L2 cache misses for *em3d*, *voronoi*, and *ft*. These are pathological cases in DDREMAP due to the padding spaces between fields [12]. STATIC shows more cache miss reductions than CMALLOC but less reductions than the two profile-based approaches, PERF_AFFI and FIELD_AFFI. PERF_AFFI and FIELD_AFFI result in more reductions in cache misses than CMALLOC. In *tsp* and 181.*mcf*, PERF_AFFI shows less reductions than FIELD_AFFI. Since PERF_AFFI takes the overhead from extra instructions into account, it, even with more cache misses, eventually performs better than FIELD_AFFI.

To investigate if the cache line size has any impact on the effectiveness of our technique, we simulated differently configured caches by changing L1D cache line size, but keeping the total cache size the same. In this experiment, we used 32 kB L1D 8-way set associative cache and 2 MB L2 8-way set associative cache. Since we only vary the line size of L1D cache, we fix the line size of the L2 cache to 64 bytes throughout this experiment. All the cache misses in Fig. 8 are normalized to the number of cache misses of ORIGINAL on 32 kB L1D with 64 byte lines and 2 MB L2 with 64 byte lines. Fig. 8a shows the normalized L1D cache miss rates as the size of L1D cache line varies from 16 to 128 byte. Fig. 8b shows their corresponding L2 cache misses in normalized form.

In general, less cache misses are likely to occur, when cache lines increase. According to our simulation results, we find similar trends across different reordering methods. Only *Ft* shows a little change across different L1D line size for most of cases except the L1D misses of ORIGINAL. PERF_AFFI achieves the best L1D and L2 combined cache performance for all different L1D line sizes. Base on these simulation results, our structure reorganization, we believe, works well across various L1D cache line sizes.

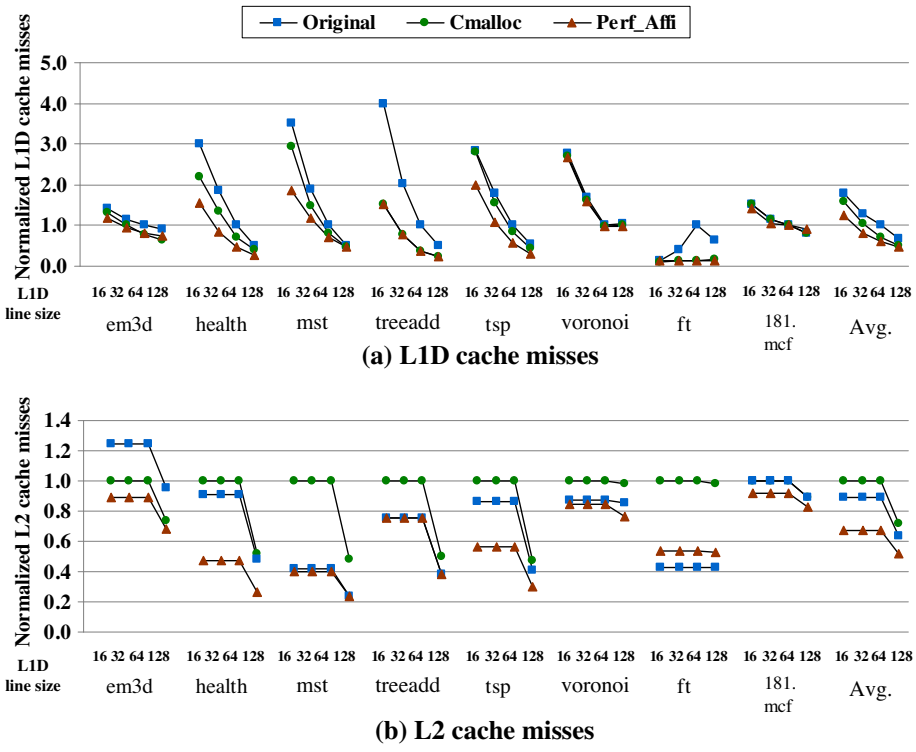


Fig. 8. Variation of cache misses in (a) L1D and (b) L2 cache as L1D cache line size varies from 16 to 128 bytes. (Normalized to the number of cache misses for Original on 8 kB L1D with 64 B lines and 512 kB L2 with 64 B lines).

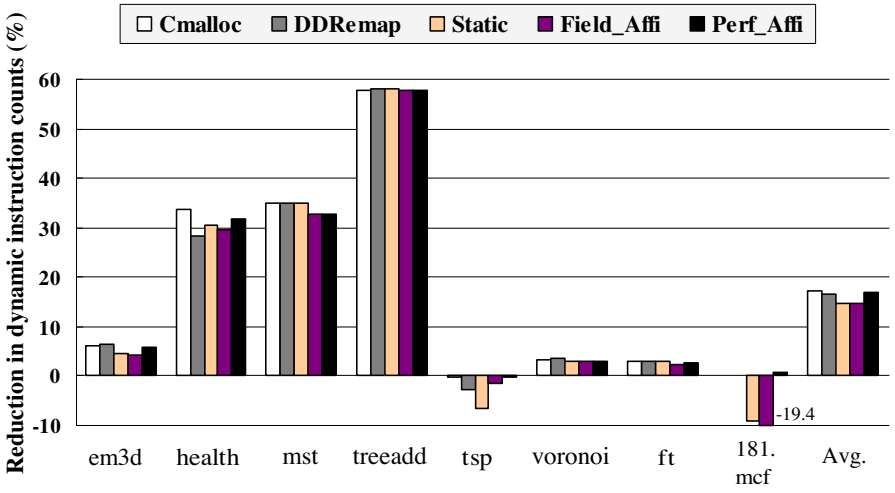


Fig. 9. Reductions in dynamic instruction counts with large inputs.

4.3. Impact on dynamic instruction counts

Fig. 9 shows the reductions in dynamic instruction counts compared to the original programs. All reorganization methods shown in the figure use pool allocation for target structures. By using custom memory management routines, we can reduce the dynamic number of executed instructions by a large amount. Since we replace `malloc()` and `free()` with the custom memory management routines, dynamic instruction counts are reduced in most of benchmarks. In some layouts, however, dynamic instruction counts are increased. This is mostly due to the extra instructions in the field accesses for the new layout.

As for Cmalloc, the average reductions are 17.3%. DDRemap, Staitc and Field_Affi reduce dynamic instruction counts by 16.5%, 14.9% and 14.8%, respectively. Reorganization methods, which particularly employs field grouping in addition to Cmalloc method, tend to execute more instructions than Cmalloc. To mitigate the penalty of the extra instructions in field grouping, Perf_Affi takes the extra instructions into account as well as the cache performance. Its reduction in the dynamic instruction count is 16.8%, which is comparable to Cmalloc.

If we take 181.*mcf* as an example, the reduction of the instruction count in Perf_Affi is much better than other layouts from Static and Field_Affi, which select the best grouping result without considering the extra instruction overheads. Our reorganization method, Perf_Affi, considers not only the cache performance but also the extra instruction overhead. Our method avoids much of the extra instruction overhead by placing the most frequently

Table 4
Reductions in execution times for the different field reorganization techniques.

Benchmarks	Input	Original (s)	Cmalloc (%)	DDRemap (%)	Static (%)	Field_Affi (%)	Perf_Affi (%)
em3d	Small	1.36	4.1	0.1	0.4	4.7	6.4
	Large	24.04	3.2	−0.1	0.1	2.7	5.9
Health	Small	2.03	19.4	13.9	20.4	21.3	22.8
	Large	26.2	5.1	7.9	12.4	11.8	14.8
mst	Small	1.60	16.9	15.5	17.9	19.6	19.6
	Large	41.78	13.0	15.0	15.9	16.5	16.5
treeadd	Small	1.49	44.4	42.6	42.6	44.4	44.4
	Large	13.2	46.3	45.2	45.2	46.3	46.3
tsp	Small	1.49	4.6	5.2	7.7	13.2	14.4
	Large	22.4	5.5	6.5	6.6	7.9	9.3
voronoi	Small	1.44	5.9	1.0	5.6	5.9	6.2
	Large	12.4	1.5	1.0	1.4	1.6	1.6
ft	Small	1.31	46.1	41.1	47.6	42.3	48.7
	Large	46.62	16.5	6.9	18.4	10.3	18.5
181.mcf	Small	7.48	0.0	8.0	3.4	4.9	10.7
	Large	107.61	0.0	17.0	14.2	16.5	18.4
Average	Small	–	17.7	15.9	18.2	19.5	21.7
	Large	–	11.4	12.4	14.3	14.2	16.4
	Both	–	14.5	14.2	16.2	16.9	19.0

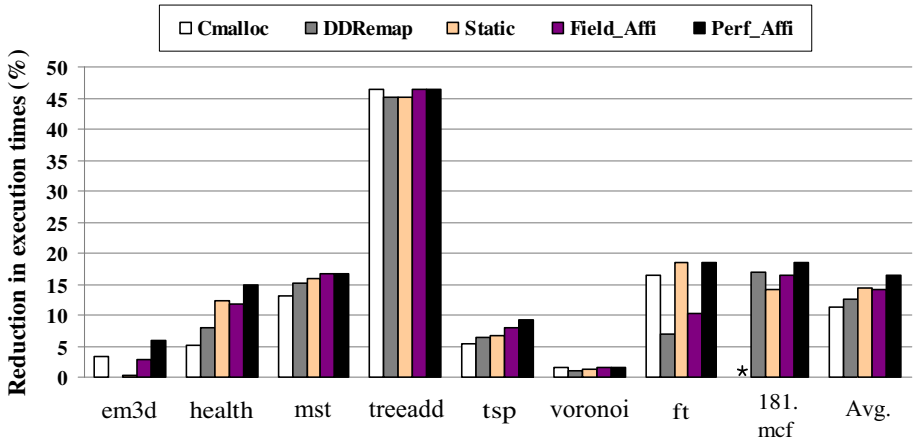


Fig. 10. Reductions in execution times with large inputs (* for 181.mcf means the pool allocation is already used in the original program.).

accessed group at the beginning, but still selecting a layout with a good cache performance.

4.4. Impact on execution times

Table 4 shows the reductions in the execution times for various layouts obtained by several reorganization methods. We perform the experiments with the two sets of inputs for each benchmark. For the profile-based methods, DDRemap, Field_Affi and Perf_Affi, the small inputs are used during the profile to determine the layout and the performance for the large inputs are measured with the same layout as the small inputs. Fig. 10 also graphically shows the reductions in execution times performed with large inputs.

The experimental results show that our Perf_Affi method achieves better performance than the other methods. Cmalloc and DDRemap reduce the execution times by 14.5% and 14.2%, respectively. Static and Field_Affi shows 16.2% and 16.9% reductions, respectively. These two methods commonly use the field affinity graph, but Static uses the static access patterns based on regular expressions, while Filed_Affi counts the neighboring accesses within the access window. Perf_Affi, our proposed method, shows 19.0% reduction, which is the best reduction overall. For all benchmarks we tested, Perf_Affi consistently performs better than other methods. The shaded numbers in Table 4 represent the best performance for the benchmark with the specified input. For all cases, Perf_Affi finds the best performance. Since our reorganization method, Perf_Affi, considers not only the cache performance but also the overheads in extra instructions, the results shown in Table 4 and Fig. 10 confirm that our method accurately estimates the actual performance differences across multiple layouts and finds the best performing layout. When other methods also find the same layout as Perf_Affi, they also achieve the same performance as Perf_Affi. For *mst*, Field_Affi finds the same layout and for *treeadd*, Cmalloc and Field_Affi find the same layout.

Compared to Static method, our profile-based Perf_Affi method improves the average performance by 3%. Improvements over Static are more noticeable for *em3d* and 181.*mcf*, which are 6% and 4% for large inputs. Even though our profile-based method does not show a large improvement over the static method, our method is still a valid approach to find the best performance for the important applications that need to be optimized to the extreme.

4.5. Cache simulation times

Cache simulation is expensive, even though we use the subset simulation for composition. When the number of fields is small,

Table 5
Time comparison between exhaustive runs and composition-based cache simulation for all feasible layouts.

Benchmarks	#Fields in target structures	Orig. exec. time (s)	Best exec. time (s)	#All feasible layouts	Estimated exhaustive run time (A) (s)	Cache simulation time (B) (s)	Ratio of B/A
em3d	7	1.4	1.3	3263	4242	526	0.12
health	3,4,7,6	2.0	1.6	3984	6374	893	0.14
mst	3,3	1.6	1.3	20	26	810	31.3
treeadd	3	1.7	0.9	10	9	568	63.3
tsp	7	1.5	1.3	3263	4242	522	0.12
voronoi	5	1.4	1.4	151	211	216	1.02
ft	5,4	1.3	0.7	188	132	43	0.32
181.mcf	15,8	7.5	6.7	8.6×10^9	5.8×10^{10}	5725	0.00

exhaustively running a program with all feasible layouts could be a lot faster than the cache simulation. Table 5 compares the total time of exhaustive running with the required time of our composition-based cache simulation. The second column shows the numbers of fields for selected target structures. For example, four structures are selected in *health* and the numbers of fields for those structures are 3, 4, 7, and 6, respectively. The third column and the forth column show the execution times of the original programs and the best execution times with the best field layouts. The fifth column is the number of all feasible layouts with the field reorganization. If multiple target structures are selected for an application, the numbers of all feasible layouts for all structures are added up. The sixth column represents the estimated execution times, when we exhaustively run the benchmark multiple times with all feasible layouts. These numbers are minimally estimated by multiplying the best execution time (the fourth column) and the number of all layouts (the fifth column). The seventh column represents the total time for our composition-based simulation, which includes the target structure identification, the trace extraction, and the cache simulation for all subsets. The eighth column shows the ratio of the composition-based cache simulation to the exhaustive run. If the ratio is less than 1, the composition-based cache simulation is faster. Otherwise, the exhaustive run is faster.

According to our experiments, our composition-based cache simulation is a faster approach to find the best performing layout for five benchmarks. For those benchmarks, our composition-base cache simulation takes only a small fraction of the time required for exhaustive runs. Exceptions are the cases in *mst*, *treeadd*, and *voroni*. For *mst* and *treeadd*, the numbers of feasible layouts are only 10 to 20. In these cases, exhaustive runs can find the best performing ones. For *voronoi*, the number of feasible layouts is 151, which results in that the exhaustive run requires almost the same as the our cache simulation method. In general, our approach is faster than exhaustive runs, if the number of fields in target structures is large or the application takes long time enough to discourage the exhaustive runs.

5. Related work

Reorganizing data has been a popular topic in the high-performance computing community. For example, high-performance fortran (HPF) compilers provide user annotation for array layout to achieve better cache behavior according to the access patterns of loops [23]. While early studies often focus on the layout of statically allocated data alone, recent studies take account of dynamically allocated data as well, since many complex applications tend to use more dynamically allocated data from the heap.

Field reorganization in structures particularly shows effectiveness in improving spatial and temporal locality by transforming the layouts of the heap-allocated structures [3–6,8–14].

Truong et al. proposed an approach to reorganize the fields of structure-type data [3]. Their method first attempts to aggregate fields from multiple structure instances with consideration of cache alignment. Using this field interleaving scheme, rarely used fields are moved away from frequently used fields.

Chilimbi et al. proposed an optimization technique that splits structures into a frequently accessed portion and a rarely accessed portion based on the profiles of field access frequencies [5]. By splitting hot fields from the whole structures they can reduce the amount of data brought into the cache memory and achieve the performance improvement on several Java applications. This is a similar effect to what Truong et al. [3] achieved using field interleaving.

Huang et al. also introduced online object reordering scheme to improve data locality of Java applications [24]. It periodically samples the currently executing method and identifies hot objects. During the copying process in the garbage collection, frequently accessed objects are placed adjacent to their related objects to increase spatial locality.

Kistler and Franz partitioned a dynamically allocated data structure to fit into a single cache line if the structure size is larger than the cache line size [6]. During partitioning, the data fields whose accesses are close together in time are placed in the same cache line to maximize data locality. Then, data fields in a cache line are ordered to minimize load latency in case of cache misses. This technique has, however, a limitation in that padding spaces are needed if a structure size is not a multiple of a cache line size.

Rabbah and Palem proposed a vertical field layout that consecutively places the same fields from multiple structure instances by using customized allocation routines and compile-time transformations of field offset calculations [9]. Their approach is similar to Truong's approach [3] in that the same fields from multiple structure instances are consecutively placed.

Zhong et al. proposed k-distance analysis to find a hierarchical partition of the program data based on their reference affinity model [10]. Reference affinity represents the degree of closeness in reference traces among a group of data. With the resulting hierarchical partition, they reorganize the whole program data by regrouping arrays and splitting structures.

Rubin et al. presented a framework that recognizes profiled access patterns with a context free grammar. If they find the same access patterns they already simulated, they reuse the previous cache simulation results to reduce the simulation time of a given access pattern for a given layout [8]. They iteratively searches all possible candidate data layouts and select the best performing one. Our approach is similar in that we also simulate all the candidate layouts, but our approach is orthogonal to theirs. Our approach reduces the cache simulation times for multiple candidate layouts by simulating primary subsets of fields and composing the simulation results from the results of the subset simulations. On the other hand, their approach reduces the cache simulation time for a data layout by reusing the simulation results in repeated access patterns.

Lattner and Adve describe an automatic pool allocation, which segregates different instances of data structures into separate memory pools [16]. They determine target structures based on the context-sensitive pointer analysis and the escape property for

the data structures. They show that their technique can allow more effective compiler optimizations and reduce the working sets of the programs, potentially improving the cache performance.

Hundt et al. develop a framework that analyzes profitability of structure layout transformation with or without profile information [11]. Their framework is capable of structure splitting, structure peeling, dead field removal, and field layout restructuring. Their optimizations are based on field affinity relations, which uses profiled or estimated field reference counts in tightly executed modules like loops, while we simulate the cache behavior with composition-based techniques and take account of extra instructions in transformed layouts.

Our paper extends our previous studies of field reorganization that colocates fields from multiple structure instances and groups closely related fields [12–14]. In our previous works, we use a field affinity graph to extract the affinities between fields. The field affinity graph associates co-access frequencies of fields with every edge in the graph. We obtain co-access frequencies of fields by profiling [12] or static approach [13,14]. Our static approach extracts run-time behaviors from source codes using Control Flow Graph (CFG) and transforms CFG into regular expressions. Regular expressions are used to represent memory access behaviors and the closures in regular expressions are handled like loops. We can build the field affinity graphs by calculating the weights of edges using the nesting depth of closures.

These graph based methods do not consider computation overhead due to the layout change and often get trapped in local optima. In this paper, we present a new profile-based structure reorganization method, which uses the composition-based cache simulation and considers computation overhead to estimate the performance of all possible layouts.

6. Conclusion

We present an advanced field reorganization method for multiple heap-allocated structures. Our field reorganization technique improves the cache performance by aggregating, compacting, and grouping fields from multiple instances of the same structure type. To find the best performing field layout, we propose a performance estimation method which uses the composition-based cache simulation along with the consideration of the extra instructions. With the resulting field layouts, our compiler automatically transforms the source programs to correctly access the fields in the reorganized layouts.

Experimental evaluation demonstrates our reorganization technique further improves the performance on top of the benefit from the pool allocation. Compared to the original programs, our field reorganization reduces the cache misses by 37% in the L1D cache and by 28% in the L2 cache. As a result, the execution time is reduced by 19%, which is an additional 5% over the pool allocation alone and an additional 3% over the static approach. We believe our field reorganization method is effective to optimize the important programs to the extreme, if they intensively use heap-allocated structures.

Acknowledgement

This work was supported by the Ministry of Knowledge Economy, Korea, under the Information Technology Research Center support program (IITA-2009-C1090-0902-0020).

References

[1] S. Carr, K.S. McKinley, C.-W. Tseng, Compiler optimizations for improving data locality, in: Proceedings of Conference on Architectural Support for Programming Language and Operating System (ASPLOS'94), 1994, pp. 252–262.
[2] M.E. Wolf, M.S. Lam, A data locality optimizing algorithm, in: Proceedings of Conference on Programming Language Design and Implementations (PLDI'91), 1991, pp. 30–44.
[3] D.N. Truong, F. Bodin, A. Seznec, Improving cache behavior of dynamically allocated data structures, in: Proceedings of Conference on Parallel Architectures and Compilation Techniques (PACT'98), 1998, pp. 322–329.
[4] T.M. Chilimbi, M.D. Hill, J.R. Larus, Cache-conscious structure layout, in: Proceedings of Conference on Programming Language Design and Implementations (PLDI'99), 1999, pp. 1–12.
[5] T.M. Chilimbi, B. Davison, J.R. Larus, Cache-conscious structure definition, in: Proceedings of Conference on Programming Language Design and Implementations (PLDI'99), 1999, pp. 13–24.
[6] T. Kistler, M. Franz, Automated data-member layout of heap objects to improve memory-hierarchy performance, ACM Transactions on Programming Languages and Systems 22 (3) (2000) 490–505.
[7] P. Panda, F. Catthoor, N. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, P. Kjeldsberg, Data and memory optimization techniques for embedded systems, ACM Transactions on Design Automation of Electronic Systems 7 (2) (2001) 149–206.
[8] S. Rubin, R. Bodik, T.M. Chilimbi, An efficient profile-analysis framework for data-layout optmizations, in: Proceedings of Symposium on Principles of Programming Languages (POPL'02), 2002, pp. 140–153.
[9] R.M. Rabbah, K.V. Palem, Data remapping for design space optimization of embedded memory systems, ACM Transactions on Embedded Computing Systems 2 (2) (2003) 186–218.
[10] Y. Zhong, M. Orlovich, X. Shen, C. Ding, Array regrouping and structure splitting using whole-program reference affinity, in: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementations (PLDI'04), 2004, pp. 255–266.
[11] R. Hundt, S. Mannarswamy, D. Chakrabarti, Practical structure layout optimization and advice, in: Proceedings of the International Symposium on Code Generation and Optimization (CGO'06), 2006.
[12] K. Shin, J. Kim, S. Kim, H. Han, Restructuring field layouts for embedded memory systems, in: Proceedings of Design Automation and Test in Europe (DATE'06), 2006, pp. 937–942.
[13] J. Jeon, K. Shin, H. Han, Layout transformations for heap objects using static access patterns, in: Proceedings of International Conference on Compiler Construction (CC'07), 2007, pp. 187–201.
[14] J. Jeon, K. Shin, H. Han, Abstracting access patterns of dynamic memory using regular expressions, ACM Transactions on Architecture and Code Optimization 5 (4) (2009) (Article 18).
[15] E. Petrank, D. Rawitz, The hardness of cache conscious data placement, in: Proceedings of Symposium on Principles of Programming Languages (POPL'02), 2002, pp. 101–112.
[16] C. Lattner, V. Adve, Automatic pool allocation: Improving performance by controlling data structure layout in the heap, in: Proceedings of Conference on Programming Language Design and Implementation (PLDI'05), Chigago, Illinois, 2005, pp. 129–142.
[17] G.C. Necula, S. McPeak, S.P. Rahul, W. Weimer, Cil: Intermediate language and tools for analysis and transformation of c programs, in: Proceedings of Conference on Compiler Construction (CC'02), 2002, pp. 213–228.
[18] Intel Corporation, IA-32 Intel Architecture Optimization Reference Manual, April 2006. Available from: <http://developer.intel.com>.
[19] Valgrind, a suite of tools for debugging and profiling linux programs, <http://www.valgrind.org/>.
[20] Olden benchmark, <http://www.cs.princeton.edu/-mcc/olden.html>.
[21] Pointer-intensive benchmark suite, <http://www.cs.wisc.edu/austin/ptr-dist.html>.
[22] Standard Performance Evaluation Corporation, SPEC CPU2000 Integer Benchmark, 2000.
[23] High Performance Fortran Forum, High Performance Fortran Language Specification, Version 1.0, Tech. Rep. CRPC-TR92225, Center for Research on Parallel Computation at Rice University, 1993.
[24] X. Huang, S. Blackburn, K. McKinley, J. Moss, Z. Wang, P. Cheng, The garbage collection advantage: Improving program locality, in: Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'04), 2004, pp. 69–80.

Keoncheol Shin received the BS and the MS degrees in Computer Science from KAIST, Korea in 2002 and 2004. He is currently a Ph.D. student at KAIST, Korea. His research interests are in the field of software optimization, in particular compiler techniques to exploit data locality in cache memory and parallelism embedded in source code. He is broadening the applications of software optimization to the parallelism in graphic applications and data manipulation in transactional memory systems.

Hwansoo Han received the BS and the MS degrees in computer engineering from Seoul National University, Korea in 1993 and 1995, and the Ph.D. degree in Computer Science from the University of Maryland at College Park in 2001. He is currently an associate professor at Sungkyunkwan University, Korea. Previously, he served as an associate professor at KAIST and as a senior engineer at Intel. His research interests include compiler technology for high-performance computing, embedded computing, and secure computing.

Kwang-Moo Choe received the BS degree in electronic engineering from Seoul National University, Korea in 1976 and the MS and PhD degrees in Computer Science from KAIST, Korea in 1978 and 1984, respectively. He is currently a professor at KAIST, Korea. Previously, he worked as a member of technical staff at AT&T Bell Labs, Murray Hill. His research interests include formal language theory, parallel evaluation of logic programs and optimizing compilers.