SCIENCE CHINA
Information Sciences

• RESEARCH PAPERS •

September 2011 Vol. 54 No. 9: 1769–1783
doi: 10.1007/s11432-011-4265-0

ASLOP: A field-access affinity-based structure data layout optimizer

YAN JiaNian[1]*, HE JiangZhou[1], CHEN WenGuang[1],
YEW Pen-Chung[2] & ZHENG WeiMin[1]

[1]*Department of Computer Science and Technology, Tsinghua University, Beijing* 100084, *China;*
[2]*Department of Computer Science and Engineering, University of Minnesota at Twin-Cities, Minneapolis, MN* 55455, *USA*

Received December 7, 2009; accepted April 15, 2010; published online May 31, 2011

Abstract By rearranging the data, data layout optimizations improve the utilization of a cache line between two of its successive refills, thus reducing the total number of cache line refills and improving the performance of a program. In this paper, we show that to enable structure data layout optimizations to be effective, two parameters, namely intra-instance affinity and inter-instance affinity, need to be considered at the same time in order to model the cache line utilization more accurately. We also propose a lightweight approach to measure intra-instance affinity and inter-instance affinity to avoid complex memory trace analyses. A prototype, called ASLOP, has been implemented in the Open64 compiler and evaluated using benchmarks from SPEC CPU 2000, SPEC CPU 2006 and Olden benchmark suites that have extensive structure types. Our approach can achieve up to 48.1% performance improvement over the original programs, and 11.9% over the optimized programs using maximal reshaping, an existing approach that is known to produce close to the best results, on the two platforms we tested.

Keywords compiler optimization, data-layout optimization, memory hierarchy, inter-instance affinity, intra-instance affinity

Citation Yan J N, He J Z, Chen W G, et al. ASLOP: A field-access affinity-based structure data layout optimizer. Sci China Inf Sci, 2011, 54: 1769–1783, doi: 10.1007/s11432-011-4265-0

1 Introduction

Data layout optimizations have been very effective in reducing cache misses and memory bandwidth requirement. By rearranging the data, data layout optimizations could improve the utilization of a cache line between two of its successive refills. As the total amount of data a program accesses during its execution is more or less fixed, improving the utilization of cache lines could thus reduce the total cache line refills and improve the overall program performance. The objective of a data layout optimization is to determine which data should be co-located in order to maximize the cache line utilization.

In many general-purpose application programs, a large amount of data is of the structure types. They are usually organized in either an array form or a pointer-chasing form. Accessing these data structure instances is a main contributor to the cache misses. The layout of these data structure instances has a

*Corresponding author (email: yanjn03@mails.tsinghua.edu.cn)


```
struct link_list {
   int x;
   int y;
   struct link_list *next;
};
struct link_list* l;
...
for (i=0; i<N; i++) {
   ....
   p=l
   while (p) {
      if (p->y>0) {
         ... =p->x;
      }
      p=p->next;
   }
}
```

Figure 1 A link list traversal example.

notable impact on the overall program performance.

Existing structure layout optimizations could be classified into three categories:

• Hotness-based approaches. Hot fields are put together in the same cache line to take advantage of their spatial locality. It avoids mixing hot fields with the cold fields in the same cache line to avoid replacing the hot fields with the cold fields, thus improving the utilization of the cache line.

• Reuse distance-based approaches. These approaches analyze reuse distances of different structure fields, and try to place the fields with similar reuse signatures [1] together.

• Maximal reshaping. This approach splits each field of a data structure into a separate new data structure. It is shown to be close to the best possible layout [2].

These approaches model data layout optimization from different perspectives, and may result in different layout plans for the same structure. For example, in the example shown in Figure 1, if half of the fields y in the list (length = L) are positive, x will be accessed $NL/2$ times and y and *next* will be accessed NL times. The hotness-based approaches will split off x from the structure *link_list* because the fields y and *next* are hot, and x is relatively cold. Reuse distance-based approaches will keep *link_list* as it is because x, y and *next* have similar reuse signatures. The maximal reshaping approach will, as always, fully split the *link_list*.

In this example, y and *next* are always accessed together and thus they should be located together. In only half of the iterations x is accessed together with the other two fields, it may be beneficial to splitting off x. However, whether it is beneficial further depends on how the list nodes are located in the memory. In a situation in which every list node being accessed is close to a list node accessed recently, splitting off x is a good strategy. Fields y and *next* in the adjacent instance should take up the space after the field x is split off, and y and *next* will be accessed with good spatial locality. This could improve the cache line utilization. In another situation in which every list node being accessed is far from the list nodes accessed recently, splitting off x will harm the performance because the data that takes up the space after the field x is split off will not be used in the near future. Cache line utilization will drop.

These observations indicate that to model the cache line utilization accurately, we must consider structure field accesses both inside a structure instance and between structure instances simultaneously.

We use intra-instance affinity and inter-instance affinity to characterize these attributes in this paper. Intuitively, intra-instance affinity indicates whether different fields in the same structure instance are referenced together. Inter-instance affinity, on the other hand, shows how close in memory space the current structure instance being accessed is to the other instances recently accessed. The formal definition of these two affinities and how their combined effects could affect the cache line utilization are discussed in section 2.

To the best of our knowledge, none of the existing approaches considers both of the affinities simultaneously when they determine the data layout. Hence, they could misjudge some optimization opportunities.

In this paper, we propose a new approach, ASLOP (Field Access **A**ffinity Based **S**tructure Data **L**ayout **OP**timization), to optimize structure data layout based on both intra- and inter-instance affinity. We make two major contributions:

• Using both inter- and intra-instance affinity to guide data layout optimizations. We consider both intra-instance affinity and inter-instance affinity simultaneously to guide data layout optimizations. We also develop a data-layout heuristic that considers the combined effect of these two attributes. It is more efficient than the existing data layout approaches.

• Lightweight approach to measure intra- and inter-instance affinity. A memory trace analysis is required to obtain an accurate measurement of intra-instance and inter-instance affinity. However, in practice, such an overhead could be prohibitively high. To address this issue, we propose a hybrid approach: using static analysis, hardware performance counters and execution frequency feedback information to

approximate intra- and inter-instance affinity.

We have implemented ASLOP in the Open64 compiler and evaluated it with 9 benchmarks from SPEC CPU 2000, SPEC CPU 2006 and Olden benchmark suites that have extensive structure types. Experiment results show that ASLOP can achieve up to 48.1% performance improvement over the original programs, and 11.9% over the optimized programs using maximal reshaping, an existing approach that is known to produce close to the best results, on the two platforms we tested.

The rest of the paper is organized as follows. Section 2 discusses the two affinities in detail. Section 3 explains the algorithm to determine a good data layout using the field access affinity attributes. Section 4 presents several key implementation issues, and section 5 shows experimental results. Related works are summarized in section 6, and section 7 concludes our work.

2 Field access affinity

In this section, we will first give the definitions of the two affinities, and then demonstrate how their combined information could lead to a good decision on the structure data layout with examples. We also introduce a more practical scheme to measure these two affinities.

2.1 Intra-instance & inter-instance affinity

Both affinities characterize the runtime access behavior to structure instances. They depend on the *context* of memory access operations and the *attributes* of the structure instances. As program behavior often repeats due to loops or recursive function calls, the two affinities usually stay the same for a certain period of time during program execution.

Intra-instance affinity. Assume a field x is being accessed. y is another field in the same structure instance accessed before x. If the number of different data accessed between accesses to x and y is less than a given number D, x and y are said to have *intra-instance affinity*.

Inter-instance affinity. Assume x is a field of structure S. A memory operation is accessing the field x of an S instance. The address of this field x is a. B is the set of addresses of the first M most recently accessed field x in other S instances. $|B| = M$. The inter-instance affinity of current memory is defined as

$$\min_{b_i \in B} |a - b_i|.$$

Parameters D and M depend on cache configuration and the structure being transformed. Intuitively, if two fields are located in the same cache line, D is the threshold that the cache line is not evicted while the two fields are being used. M is related with cache line size, the field size, etc. It is the threshold that if the structure was split, two fields from different instances could be located into one cache line.

2.2 The effect of the two affinities

Each memory access pattern has its own preferable data layout that could optimize the cache performance. A data layout algorithm should estimate the effect of a data layout strategy over all memory accesses, and decides its best data layout based on such estimations. Memory accesses in a repeated code region, for example a loop or a recursive function, usually share the same access pattern.

For simplicity, we only include two fields in the structure of our example. The structure instances are organized as an array. Assume there is no hardware prefetching, and the size of a cache line is 64-byte. The program example is shown in Figure 2. There are two possible layouts for the structure *dpair*. They are shown in Figure 3(a) and (b).

There are 6 memory access patterns in the example program that represent the memory accesses in the 6 loops respectively. The patterns are listed in Table 1.

The first pattern has no preference to either of the layouts. Both of them would get 100% cache line utilization. The second pattern prefers layout one whose cache line utilization is 25%, while layout two would yield only 12.5%. For example, in the first two iteration of loop 2, accessing $arr[0].x$, $arr[0].y$,

```
struct dpair {
        double x;
        double y;
} arr [M];
.....
for (i=0; i<I1; i++)  {
        sum += arr [i]. x;
        sum += arr [i]. y;
}
for (i=0; i<I2; i+=8)  {
        sum += arr [i]. x;
        sum += arr [i]. y;
}
for (i=0; i<I3; i++)  {
        sum += arr [i]. x;
}
for (i=0; i<I4; i+=8)  {
        sum += arr [i]. x;
}
for (i=0; i<I5; i++)  {
        sum += arr [i]. y;
}
for (i=0; i<I6; i+=8)  {
        sum += arr [i]. y;
}
```

Figure 2 Different field access patterns of a two-field structure.

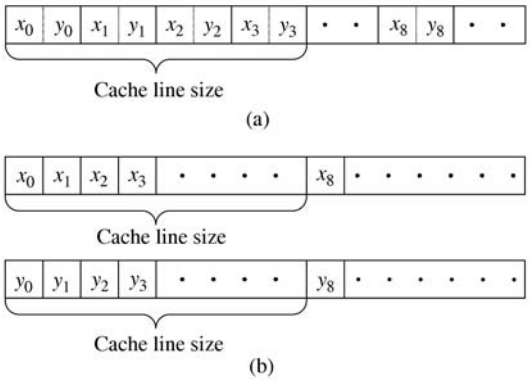

Figure 3 Two possible layouts for the array of a two-field structure in Figure 2. (a) Layout one: two fields are conjunctive; (b) layout two: two fields are split.

Table 1 Memory access patterns of example program

Code portion	Intra-instance affinity	Inter-instance affinity
Loop 1	Yes	16 bytes
Loop 2	Yes	128 bytes
Loop 3	No, only x accessed	16 bytes
Loop 4	No, only x accessed	128 bytes
Loop 5	No, only y accessed	16 bytes
Loop 6	No, only y accessed	128 bytes

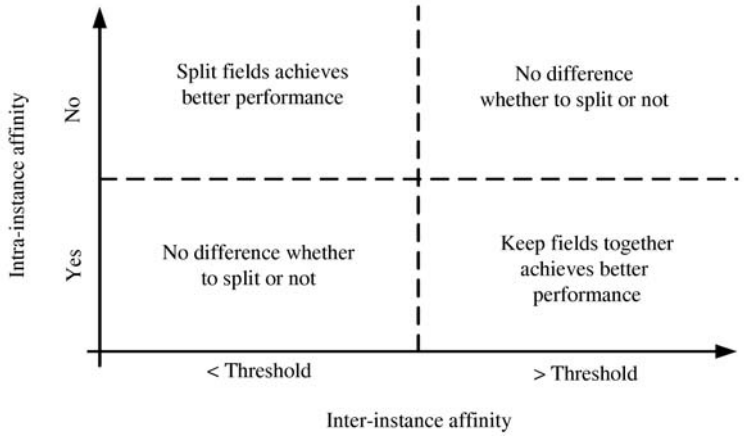

Figure 4 Data layout decision depends both on intra-instance and inter-instance affinity.

$arr[8].x$, $arr[8].y$ will need to bring in two cache lines if x and y are located together as in layout one. Using layout two, it will need to bring in four cache lines. The third and fifth patterns prefer layout two, which will get 100% cache line utilization, instead of just 50%. The fourth and sixth patterns have no preference because both of the layouts will get the same 12.5% utilization.

If the number of memory accesses in loop 2 is larger than that in loops 3 and 5, the optimal layout strategy will be to keep x and y together, otherwise, we should split x and y.

In this example, 128-byte is the threshold for inter-instance affinity. If the inter-instance affinity of a field access is smaller than 128-byte, by splitting the fields, we can put this field and the prior field into the same cache line to improve spatial locality. The relationship between data layout decision and the combined affinities can be summarized as in Figure 4.

2.3 Measurement of intra-instance and inter-instance affinity

In general, precisely measuring the two affinities requires a detailed analysis on memory traces whose overhead could be prohibitively high in practice. Our goal is to have a simplified estimate with an overhead at the level of current profile-based optimizations, which is about 3X-5X slowdown. Our approach gives up measuring parameters D and M to decide intra- and inter-instance affinity. Instead, we use heuristic to detect whether memory references have intra-instance affinity and whether their inter-instance affinity is larger or smaller than the threshold, which are directly used in deriving data layout strategy.

2.3.1 *Measurement of intra-instance affinity*

Intra-instance affinity can be measured by analyzing the source code. If the accesses of the two fields in the same structure instance are close in the source code, the two fields probably have intra-instance affinity, unless there are many other memory accesses between the two.

When measuring intra-instance affinity, a function is viewed as being composed of several *scopes* textually. A *scope* is the largest span of a code region that does not contain loops and function calls. We assume the number of other different memory accesses between a pair of memory accesses is smaller than D as defined in subsection 2.1 for the intra-instance affinity.

Branches will introduce different execution paths into a scope, and intra-instance affinity in different execution paths may differ. If the number of branches in a scope is n, the number of different execution paths would be 2^n, which is too complicated to analyze each execution path.

For simplicity, in the source code, if two fields of the same structure instance are accessed within a scope, these two fields are considered to have intra-instance affinity. The number of runtime memory accesses in which the two fields have such intra-instance affinity is assigned to the smaller execution count of the access to each of the two fields.

2.3.2 *Measurement of inter-instance affinity*

Different from intra-instance affinity, runtime memory accesses originating from the same memory operation in the source code may have different inter-instance affinities. For simplicity, we average the inter-instance affinity of the runtime memory accesses originating from the same memory operation in the source code as an approximation.

As indicated in subsection 2.2, depending on whether the inter-instance affinity is larger or smaller than a certain threshold, a memory operation may have different preferences for the data layout. The threshold is a distance in memory space that if the distance of two structure instances is smaller than the threshold, after splitting, the fields in the two instances will be located in the same cache line.

Therefore, the objective of measuring inter-instance affinity and comparing it with the threshold to guide data layout can be approximated by measuring the cache miss rate of the program in which all structure instances are fully split.

However, using only L1 or L2 cache miss rate is not sufficient. Hardware prefetching will eliminate cache misses of memory access sequence that has constant stride. However, the inter-instance affinity is irrelevant memory access stride. In ASLOP, $cache\_miss_{\mathrm{L2}}/cache\_miss_{\mathrm{L1}}$ is used to measure inter-instance affinity, as it has a positive correlation with the average inter-instance affinity. The inference is as follows:

Assume in an ideal program, all memory operations are accessing the instances of structure S. $\{v_0, v_1, \ldots, v_{N-1}\}$ is the accessing sequence. $fld(v_i)$ is the field of some S's instance that v_i accesses. Each v_i belongs to an equivalence class, $equ(v_i)$. Two accesses v_i and v_j belong to the same equivalence class, if and only if $fld(v_i)$ and $fld(v_j)$ are in the same cache line when they are both in cache.

For an access v_i, we have a sequence $\{u_0, u_1, \ldots, u_{M-1}\}$ composed of non-duplicating elements in the sequence $\{equ(v_{i-1}), equ(v_{i-2}), \ldots, equ(v_0)\}$, leaving only the first appearance of each equivalence class. Assume conditional probability $P(u_j = equ(v_i) | equ(v_i) \notin \{u_0, u_1, \ldots, u_{j-1}\}) = \lambda$, where λ is a value dependent on the average inter-instance affinity for memory accesses originating from the same memory

operation in the source code. It has a negative correlation with the average inter-instance affinity. The smaller the affinity is, the more possible it is that data in the same cache line are accessed together.

By the definition of λ, we have the following probability:

$$P(u_j = equ(v_i)) = \lambda(1-\lambda)^j.$$

For simplicity, we assume that the cache is fully-associative. However, for set-associative caches, the conclusion keeps its monotonicity. For a cache with L cache lines, let $p(L)$ be the probability of a cache miss for v_i.

$$p(L) = \sum_{j=L}^{\infty} P(u_j = equ(v_i)) = (1-\lambda)^L.$$

Let L_1 and L_2 be the number of cache lines of L1 cache and L2 cache respectively, we have

$$\frac{p(L_2)}{p(L_1)} = (1-\lambda)^{L_2-L_1}.$$

Since $p(L_1)$ is the L1 cache miss probability, $p(L_2)$ is the L2 cache miss probability, statistically $cache\_miss_{\mathrm{L2}}/cache\_miss_{\mathrm{L1}}$ is efficient to approaching $(1-\lambda)^{L_2-L_1}$, which is monotonically decreasing with λ. As is introduced earlier, λ has a negative correlation with the average inter-instance affinity. $cache\_miss_{\mathrm{L2}}/cache\_miss_{\mathrm{L1}}$ has monotonicity with the average inter-instance affinity and it is sufficient for measuring requirements of structure data-layout optimization.

Hardware performance counters are employed to measure the L1 and L2 cache miss rates. Hardware performance counters could report the cache miss counts of the L1 and L2 caches and the addresses of the instructions that trigger them. With the help of the debugging information, these counts can be traced back to the source code.

The inter-instance threshold depends on the structure field size, and it would change if a layout strategy puts fields together. For simplicity, we approximate the threshold with a fixed value, and get this value from a micro-benchmark that accesses the array of a structure with a certain stride plus a small random offset.

3 Algorithm

Data layout optimization has been proved to be an NPC problem [3]. The algorithm proposed in this section is trying to find a data-layout strategy that eliminates cache line refills as much as possible, with acceptable time and space complexity.

We first quantify the effect of the basic decision in structure data layout optimization, which is whether to put two fields together or separate them, with an ideal example. Then, we extend the result to general cases and give our data-layout optimization algorithm.

3.1 An ideal example

Assume that a program, P, has only one data object arr. It is an array of a two-field-structure, and the fields are $fld1$ and $fld2$. All of the memory operations in P are accessing arr. s_1 is the size of $fld1$, s_2 is the size of $fld2$, s_c is the cache line size. $s_1 < s_c, s_2 < s_c$. The data references can be divided into 6 categories by considering both intra-instance affinity and inter-instance affinity. The total reference count of each category is listed as follows:

$\bar{\beta}_{1,2}$ = smaller than threshold, have intra-instance affinity,

$\ddot{\beta}_{1,2}$ = larger than threshold, have intra-instance affinity,

$\bar{\theta}_{1,\neg 2}$ = smaller than threshold, only access $fld1$,

$\ddot{\theta}_{1,\neg 2}$ = larger than threshold, only access $fld1$,

$\bar{\theta}_{2,\neg 1}$ = smaller than threshold, only access $fld2$,

$$\ddot{\theta}_{2,\neg 1} = \text{larger than threshold, only access } fld2.$$

For example, in the code example in Figure 2, these reference counts are

$$\bar{\beta}_{1,2} = I1, \quad \bar{\theta}_{1,\neg 2} = I3, \quad \bar{\theta}_{2,\neg 1} = I5,$$
$$\ddot{\beta}_{1,2} = I2, \quad \ddot{\theta}_{1,\neg 2} = \frac{1}{8}I4, \quad \ddot{\theta}_{2,\neg 1} = \frac{1}{8}I6.$$

There are two layout options as shown in Figure 3(a) and (b), except the field names are different.

For simplicity, we again assume that the cache is fully-associative. However, the conclusion is again applicable to set-associative caches. In layout one, the number of cache line refills is

$$CR_1 = \frac{\bar{\beta}_{1,2}(s_1+s_2)}{s_c} + \ddot{\beta}_{1,2} + \frac{\bar{\theta}_{1,\neg 2}(s_1+s_2)}{s_c} + \ddot{\theta}_{1,\neg 2} + \frac{\bar{\theta}_{2,\neg 1}(s_1+s_2)}{s_c} + \ddot{\theta}_{2,\neg 1} + f(\bar{\beta}_{1,2}) + f(\bar{\theta}_{1,\neg 2}) + f(\bar{\theta}_{2,\neg 1}). \tag{1}$$

In eq. (1), it is assumed that if the inter-instance affinity is smaller than the threshold, the instances recently accessed are adjacent to each other. Function $f(x)$ is used to revise the error by using such assumption.

For simplicity, we could ignore such error. As will be explained later, this approximation is sufficiently accurate for the purpose of deriving a data-layout plan. The number of cache line refills becomes

$$CR_1' = \frac{\bar{\beta}_{1,2}(s_1+s_2)}{s_c} + \ddot{\beta}_{1,2} + \frac{\bar{\theta}_{1,\neg 2}(s_1+s_2)}{s_c} + \ddot{\theta}_{1,\neg 2} + \frac{\bar{\theta}_{2,\neg 1}(s_1+s_2)}{s_c} + \ddot{\theta}_{2,\neg 1}. \tag{2}$$

Similarly, in layout two, the number of cache line refills is

$$CR_2 = \frac{(\bar{\beta}_{1,2}+\bar{\theta}_{1,\neg 2})s_1}{s_c} + \ddot{\theta}_{1,\neg 2} + \frac{(\bar{\beta}_{1,2}+\bar{\theta}_{2,\neg 1})s_2}{s_c} + \ddot{\theta}_{2,\neg 1} + 2\ddot{\beta}_{1,2}. \tag{3}$$

Subtracting eq. (2) from eq. (3), we get

$$\Delta_{CR} = CR_2 - CR_1' = \ddot{\beta}_{1,2} - \frac{s_2\bar{\theta}_{1,\neg 2}}{s_c} - \frac{s_1\bar{\theta}_{2,\neg 1}}{s_c}. \tag{4}$$

If the $\Delta_{CR} < 0$, it is profitable to choose layout two, otherwise, layout one.

3.2 Data layout algorithm

Based on the discussion in subsection 3.1, we can derive A structure data layout algorithm for structures that have multiple fields. For any pair of fields i and j, we could divide the accesses into six categories considering both intra-instance and inter-instance affinity. The references count of each category is listed as follows:

$\bar{\beta}_{i,j}$ = smaller than threshold, access both the fields,
$\ddot{\beta}_{i,j}$ = larger than threshold, access both the fields,
$\bar{\theta}_{i,\neg j}$ = smaller than threshold, only access i,
$\ddot{\theta}_{i,\neg j}$ = larger than threshold, only access i,
$\bar{\theta}_{j,\neg i}$ = smaller than threshold, only access j,
$\ddot{\theta}_{j,\neg i}$ = larger than threshold, only access j.

With some approximation, the problem of finding a data-layout plan for a multi-field structure can be divided into several two-field-structure problems. The data-layout algorithm is given in Algorithm 1. S_f is the field set of the target structure. s_i is the size of field i, where $i \in S_f$. $\bar{\beta}_{i,j}$, $\ddot{\beta}_{i,j}$, $\bar{\theta}_{i,\neg j}$, $\ddot{\theta}_{i,\neg j}$, $\bar{\theta}_{j,\neg i}$, $\ddot{\theta}_{j,\neg i}$ $\forall i,j \in S_f$ are all referred to as $RefCnts$ for simplicity. The algorithm computes Δ_{CR} for each pair of fields in the current field set S_f' to find $\Delta_{\max}$, the maximal Δ_{CR}. Meanwhile, the fields yielding $\Delta_{\max}$

Algorithm 1 Compute data-layout plan.

input: S_f, *RefCnts*, s_i, $i \in S_f$
output: data-layout plan S_f'

```
1  S_f' ← S_f
2  Δ_max ← 0
3  repeat
       Δ_max ← max_{∀i,j∈S_f'} (Δ_CR(i, j));
4
5      if Δ_max > 0 then
6          (i, j) ← Fields(Δ_max);
7          k ← MergeField(i, j, RefCnts);
8          s_k = AlignAdjust(s_i + s_j);
9          Remove i, j from S_f';
10         Add k to S_f';
11     end
12 until Δ_max ≤ 0;
13 return (S_f');
```

are recorded. If there is only one field in S'_f, $\Delta_{\max}$ is zero. If the $\Delta_{\max}$ is positive, the pair of the fields will be merged. The merged field will be considered as one field in the next iteration of the algorithm. s_i, $i \in S_f$ and $RefCnts$ are updated accordingly. The algorithm iterates until no positive $\Delta_{\max}$ is found. Function $Fields()$ returns the fields of $\Delta_{\max}$. Function $MergeField()$ merges two fields and updates $RefCnts$ accordingly. The details of $MergeField()$ are described later. $AlignAdjust()$ adjusts the size of the merged field according to field alignment requirements. Set S'_f contains the final data layout plan when the algorithm stops. Original fields that are merged in S'_f will be co-located. Different fields in S'_f will be separated.

Updating $RefCnts$ accurately after each field merging would introduce too much complexity in both computation and storage. Assume that $\bar{B}_s$, $s \in 2^{S_f}$ (where 2^{S_f} is the power set of S_f) is the total number of field accesses, any field in set s has intra-instance affinity with the field being accessed, and the access's inter-instance affinity is smaller than the threshold. $\ddot{B}_s$ is similarly defined, except that the access's inter-instance affinity is larger than the threshold. The accurate way to update $RefCnts$ is

$$\bar{\beta}_{i,j} = \sum_{\forall s \in 2^{S_f}, \{i,j\} \subseteq s} \bar{B}_s, \quad \bar{\theta}_{i,\neg j} = \sum_{\forall s \in 2^{S_f}, \{i\} \subseteq s, \{j\} \nsubseteq s} \bar{B}_s,$$
$$\ddot{\beta}_{i,j} = \sum_{\forall s \in 2^{S_f}, \{i,j\} \subseteq s} \ddot{B}_s, \quad \ddot{\theta}_{i,\neg j} = \sum_{\forall s \in 2^{S_f}, \{i\} \subseteq s, \{j\} \nsubseteq s} \ddot{B}_s,$$

i and j are the fields in S'_f after each field merging. However, the numbers of distinct $\bar{B}_s$ can be as many as $2^{|S_f|}$ and so is $\ddot{B}_s$. This complexity is far from acceptable.

Function $MergeField()$ makes some approximation in computing $RefCnts$. Assume fields i, j are the fields being merged, the new field is called k, and $m \in S'_f, m \neq i, m \neq j$. As illustrated in Figure 5, points in the three circles represent field accesses to one of the three fields. Points in the overlap area of the circles represent fields that have intra-instance affinity in their accesses. For example, for accesses in area D, fields i and j have intra-instance affinity. And in area G, each pair of the three fields have intra-instance affinity. Let $|A|$ denote the number of access in area A. After merging, the new $RefCnts$ are

$$\bar{\beta}_{k,m} = |E| + |F| + |G| = \bar{\beta}_{i,m} + \bar{\beta}_{j,m} - |G|,$$
$$\bar{\theta}_{k,\neg m} = |A| + |B| + |D| = \bar{\theta}_{i,\neg m} + \bar{\theta}_{j,\neg m} - \bar{\beta}_{i,j} + |G|,$$
$$\bar{\theta}_{m,\neg k} = |C| = \bar{\theta}_{m,\neg i} - \bar{\beta}_{j,m} + |G|.$$

In the above equations, only $|G|$ cannot be computed from $RefCnts$ before the field merges. We app-

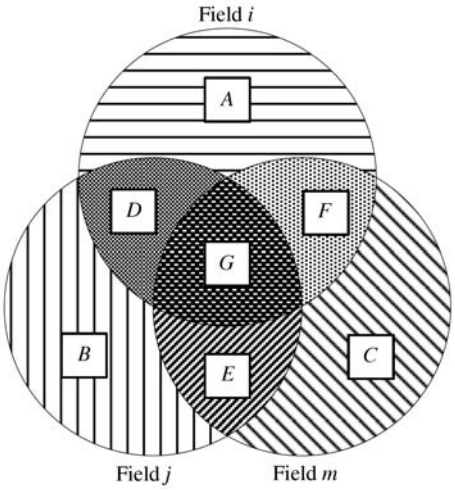

Figure 5 Illustration of fields relationship in field merging.

roximately compute $|G|$ by

$$|G| = \frac{1}{3}\left(\bar{\beta}_{i,j}\frac{\bar{\beta}_{j,m}}{\bar{\theta}_{j,\neg m}+\bar{\beta}_{j,m}} + \bar{\beta}_{i,m}\frac{\bar{\beta}_{i,j}}{\bar{\theta}_{i,\neg j}+\bar{\beta}_{i,j}} + \bar{\beta}_{j,m}\frac{\bar{\beta}_{i,m}}{\bar{\theta}_{m,\neg i}+\bar{\beta}_{i,m}}\right),$$

$\ddot{\beta}_{k,m}$, $\ddot{\theta}_{k,\neg m}$ and $\ddot{\theta}_{m,\neg k}$ can be computed similarly.

Eq. (4) has an assumption that the merged field's size is smaller than the cache-line size. Theoretically, the size of the merged field may exceed cache-line size. However, in practice, it is not common to have such a large merged field. None of the benchmark studied has a merged field larger than 64 bytes.

For guiding data-layout, only whether the Δ_{CM} is positive matters. Experiment result shows that with all the approximations we have made, our approach keeps good balance between usability and effectiveness. Comparison between our work and maximal reshaping can be found in section 5.

The complexity of Algorithm 1 is $O(n^2)$ where n is the number fields in the structure. In practice, the number of fields is usually not too large and a complexity of $O(n^2)$ is acceptable.

4 Implementation

We have implemented a prototype, ASLOP, in the Open64 compiler [4]. Its framework is shown in Figure 6. FDO FB is Open64's instrumentation-based feedback module, from which ASLOP obtains BB (basic block) frequencies. PMU FB is the feedback module from hardware performance counters, from which ASLOP computes inter-instance affinity.

Implementation of ASLOP in Open64 has three phases.

• Local phase: Source code is processed procedure-by-procedure to collect parameters for layout algorithms and to perform data-layout safety checks.

• Global phase: Layout decisions are made by the layout algorithms based on the parameters collected in the local phase.

• Transformation phase: The layout decisions are applied to structures by transforming both structure definitions and references in the code.

In the rest of this section, we introduce four implementation issues briefly.

4.1 *RefCnts* collection

As stated in subsection 2.3, if two fields i and j are both accessed within a scope, they are considered to have intra-instance affinity. The number of runtime memory accesses in which the two fields have such intra-instance affinity is assigned to the smaller one of the execution counts of the source code that accesses i and j. If the inter-instance affinity of these memory access is smaller than the threshold, the

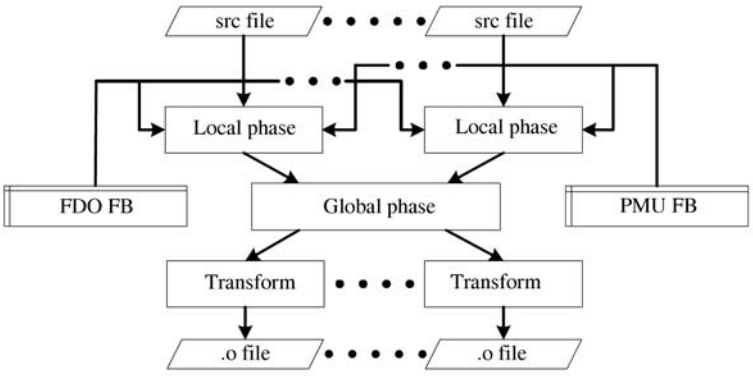

Figure 6 ASLOP framework.

number of these accesses is added to $\bar{\beta}_{i,j}$, otherwise, added to $\ddot{\beta}_{i,j}$. If a field k is not accessed in the scope, the number of these accesses is also added to $\bar{\theta}_{i,\neg k}$ or $\ddot{\theta}_{i,\neg k}$ accordingly.

4.2 Safety check

Previous works on data layout either targeted type-safe programming languages or applied conserve restriction on C/C++ to guarantee the correctness of data layout transformations. ASLOP belongs to the latter one. If one of the following conditions is true for a structure type, ASLOP stops performing data-layout transformations on it. This restriction is comparable with the one used in [5].

• There is a type cast from or to a structure type. Casting from *void* * in a dynamic memory allocation is an exception.

• A structure pointer is passed to a function whose source code is not available. Known safe standard library calls are exceptions, e.g. printf(), etc.

• The pointer of a structure type is passed to an indirect function call.

• A structure type has bit fields.

• A structure type has global instances, and the full call graph is not available in the global phase. Calls to libraries that are known to be safe are allowed.

• A structure type is nested in a union type.

4.3 Memory pools

Memory pool places instances of the same structure type in adjacent memory locations, and avoid mixing them with the data of other types. This enables ASLOP to use a uniform way to handle structure instances that are organized in either a pointer-chasing form or an array form.

In ASLOP, each part of a split structure has its own pool. A pointer variable pointing to a structure instance before applying data layout transformation becomes an index that is used in addressing all split parts. For example, assume structure *link_list* in Figure 1 is split into two parts. Figure 7 shows the data-layout change using memory pools. Figure 8 shows the change in the source code. Note that the point p in the transformed code is acting as a subscript.

4.4 Transformation

ASLOP performs transformation in two steps, generating new types and transforming the intermediate representation (IR) of the source code.

In the first step, for each split structure, T, ASLOP creates new types for each of its split parts, and changes the data types of the corresponding variables. Any variable with type T, $T[\,]$ or $T*$ is transformed to an index to the memory pool. If there are more than one variable with type T or $T[\,]$ in the same scope, only one index variable is created for them because the indices of others can be obtained from the first index and a constant offset. The fields of structures with type T, $T[\,]$ or $T*$ also need such transformations.

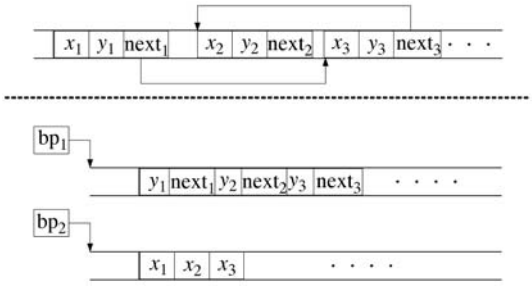

Figure 7 Layout demonstration of using memory pool.

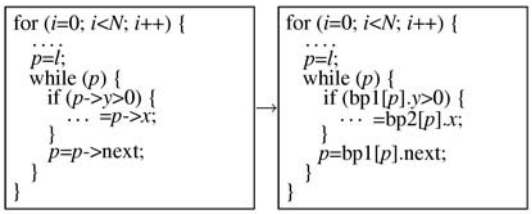

Figure 8 Demonstration of program transformation.

The IR will be transformed in the next step.

• Load and store: Loads and stores for structure fields are transformed to indirect references by adding memory pool base address, split part offset, and the new field offset.

• Pointer arithmetic: An addition or subtraction between a pointer and an integer will be transformed into an operation between an index and an integer, and a subtraction between pointers will be transformed into a subtraction between indices.

• Structure assignments: A structure instance assignment will be transformed into multiple assignments each for a split part. If the split structure is nested in other structures, instance assignments of these structures should be transformed too.

• Memory pool initialization: ASLOP initializes memory at the beginning of main().

• Memory allocation and deallocation: For split structures, ASLOP changes the dynamic allocation and deallocation function calls with memory pool's interface. For local instance, ASLOP generates code to allocate at the beginning of the function and deallocate before the function exits. For global instance, allocation is done at the beginning of main(), and deallocate before program exits. Besides, for those structures with fields of type T or $T[\ ]$, interface for calls to memory pool is also needed for memory allocation and deallocation.

5 Evaluation

5.1 Methodology

The primary objective of data-layout optimizations is to make good use of the cache hierarchy. As we described in the previous sections, our data-layout approach requires some architecture dependent parameters. We choose two platforms, Aries and Virgo, whose cache configurations are very different, to evaluate ASLOP. Their configurations are listed in Table 2. The hardware prefetch mechanism on these two platforms are turned on by default.

Totally nine benchmarks are used in our evaluation. 179.art and 181.mcf are from SPEC CPU2000 [6]; 462.libquantum, 429.mcf and 472.moldyn[1)] from SPEC CPU2006 [7]; em3d, health, treeadd and tsp from Olden benchmarks [8]. The Olden benchmarks provide a common base of reference with previous works on structure data-layout [1, 2, 9–13], while others from standard benchmark suites provide insight into complicated programs. Other benchmarks in SPEC CPU suits fail in legality test that ASLOP would not perform data layout transformation on them.

It should be noted that although 429.mcf and 181.mcf share the same core algorithm, they have different structure definitions which have a significant impact on data-layout optimizations. We treat them as two separate applications.

For benchmarks from SPEC CPU suite, we use SPEC's *train* data set to get runtime feedback, and use *ref* data set for performance evaluation. For benchmarks from Olden suite, the difference between the training data set and the testing data set can be found in Table 3. The table also gives the memory footprint when running the testing data set on each benchmark.

1) 472.moldyn was a candidate of SPEC CPU2006. Its source code can be found in SPEC CPU2006 V1.0.

Table 2 Experimental platform configuration

Platform	CPU	L1D cache per core	L2 cache per core	L3 cache
Aries	Opteron 8214	64 KB	1024 KB	None
Virgo	Xeon E5504	32 KB	256 KB	4 MB

Table 3 Benchmark characteristics

Benchmark	Training input	Testing input	Memory footprint
179.art	train	ref	4140 K
181.mcf	train	ref	112 M
429.mcf	train	ref	1.6 G
462.libquantum	train	ref	95 M
472.moldyn	train	ref	105 M
em3d	12000 600 75 1	24000 1200 75 1	1.3 G
health	4 3500 7	4 7000 7	19 M
treeadd	22 1	24 1	512 M
tsp	3276800	16384000	1.0 G

One recent data-layout work [1] studied various existing data-layout optimizations. It showed that *maximal reshaping* achieves the best or close-to-the-best performance improvement in the benchmarks studied. In this paper, we also use *maximal reshaping* as a baseline for our comparison

5.2 Performance evaluation

The benchmarks are compiled with -O3 -ipa. Software prefetch in Open64 is enabled. We run each benchmark 6 times and take the average value. The normalized execution time is shown in Figure 9. The term *Original* represents benchmarks without any data-layout transformation, and *MaxRes* for benchmarks that applies maximal reshaping, *ASLOP* for benchmarks that applies ASLOP transformation and *Pool* for benchmarks that only applies memory pooling.

Compared with the native compiler optimizations without our data-layout optimizations enabled, ASLOP improves applications' performance dramatically. Among the totally 18 testings (9 benchmarks $\times$ two testing platforms), ASLOP achieves more than 20% performance improvement in 13 of them. In all 18 cases, ASLOP has a significant performance gain over the original program. On average, ASLOP gets 48.1% and 27.7% performance improvement on the two platforms, respectively.

ASLOP is also significantly better than *maximal reshaping*. In 4 out of 9 benchmark applications, as shown in Table 4, ASLOP suggests data layouts different from the ones in maximal reshaping. All of them result in better performance. On average, ASLOP is 9.5% and 11.9% faster than maximal reshaping on the two platforms, respectively.

The major performance improvement of ASLOP is from better data layouts than *Pool*.

Figure 10 shows the normalized last-level cache miss rates, which could explain the cause of the performance differences in Figure 9.

ASLOP also shows consistent performance improvement on the two platforms which have very different cache configurations. It indicates that our approach is applicable to different platforms.

Table 4 shows the data layouts ASLOP suggests. These structures are proved by ASLOP to be both safe and beneficial to transform. In the table, we number the fields according to their definition orders. For example, the first field in a structure is numbered 0, the second 1, etc. Fields within the same bracket are located together in the suggested layout. Fields in different brackets are separated.

5.3 Overhead evaluation

Applying ASLOP requires BB (basic block) execution frequency and PMU counter feedback. The former is collected through running instrumented programs. The latter is collected in PMU sampling runs.

Table 5 shows the overhead of applying ASLOP to the benchmarks studied. The *Instrument* column lists the slowdown of running instrumented programs, comparing with original version. The *PMU*

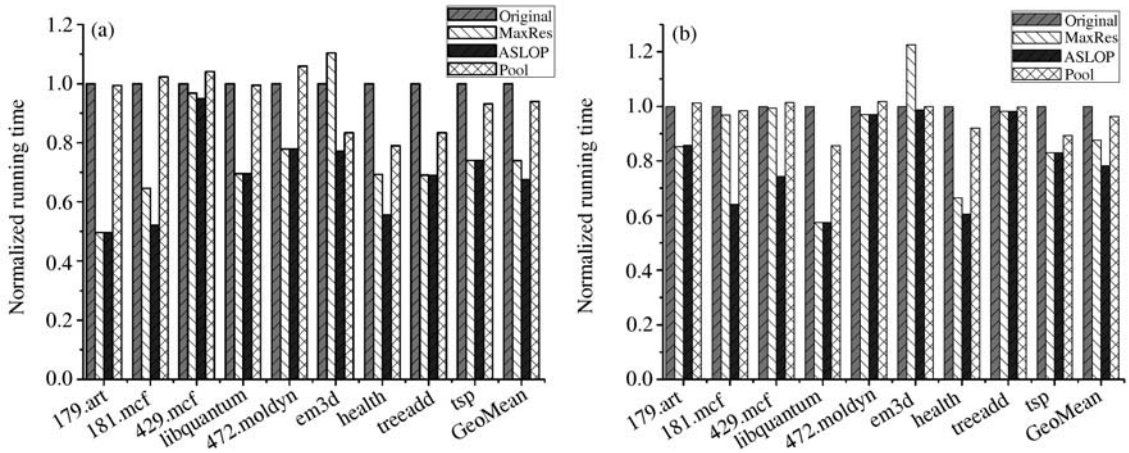

Figure 9 The normalized execution time compared to benchmarks without data-layout transformation. (a) Testing platform: Aries; (b) testing platform: Virgo.

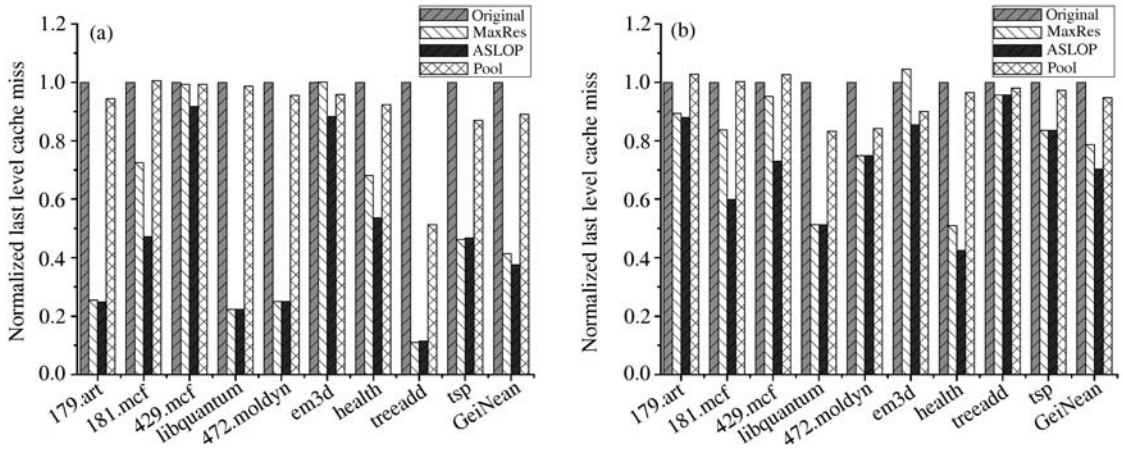

Figure 10 The normalized last level cache miss compared to benchmarks without data-layout transformation. (a) Testing platform: Aries; (b) testing platform: Virgo.

Table 4 Structure data layouts

Benchmarks	Strategy
179.art	f1_neuron{(0),(1),(2),(3),(4),(5),(6),(7)}, xyz{(0),(1)}
181.mcf	node{(2 3 7 8 4),(6 12),(0),(1),(5),(9),(10),(11),(13)}
429.mcf	node{(0 2 6),(1 3 4 11),(9 13),(5),(7),(8),(10),(12)}
libquantum	quantum_reg_node_struct{(0),(1)}
472.moldyn	Mol{(0),(1),(2)}
em3d	node_t{(3 4 6),(0),(1),(2),(5)}
health	List{(0 1),(2)}, Patient{(0),(1 2),(3)}
treeadd	tree{(0),(1),(2)}
tsp	tree{(0),(1),(2),(3),(4),(5),(6)}

sampling column lists the slowdown of sampling runs. The last column lists the additional storage requirements.

The slowdown from instrumentation and sampling varies. On average, it needs about 4.2X the program execution time to collect profiled data. The overhead of storage requirement is very small and can be ignored. ASLOP is quite efficient compared with memory trace-based approaches.

Table 5 Overhead of applying ASLOP

Benchmark	Normalized runtime		Storage
	Instrument	PMU sampling	(KB)
179.art	1.228	1.081	19.0
181.mcf	2.441	1.103	20.3
429.mcf	2.004	1.054	21.5
462.libquantum	6.379	1.713	32.9
472.moldyn	3.940	1.119	14.2
em3d	4.066	1.015	10.6
health	1.079	1.059	8.2
treeadd	7.922	1.032	2.1
tsp	5.383	1.015	7.9
Average	3.123†	1.117†	15.2‡

† Geometric mean; ‡ Arithmetic mean.

6 Related work

Cache optimizations have been studied extensively. There are a lot of works targeting scientific, array-based programs. These optimizations improve program's locality through loop nest transformations and achieve significant performance improvements. However, they are not applicable to general purpose programs which usually have complex control flow and pointer chasing data structures.

Truong et al. [9] proposed *instance interleaving* for layout data organized in structure type. They add padding arrays in a structure's declaration that could divide the structure into several parts. They used a customized memory allocator that interleaves different instances. However, their work did not have an algorithm to decide how many portions to divide a structure into and how to rearrange the fields among these portions. Instead, their method arranges the fields according to its hotness.

Chilimbi et al. [14] and Hundt et al. [4] give a data-layout method that split a structure into a hot portion and a cold portion. A pointer is inserted into the hot portion pointing to the cold one. This two-portion scheme restricts layout strategies that can be applied, thus it loses certain layout opportunities. Hundt [5] also proposes *structure peeling*, which is similar to maximal reshaping discussed earlier.

Chilimbi [14], Hundt [15] and Kistler et al. [11] proposed field reordering for large data structures. Differently, Chilimbi [14] and Hundt [5] use field affinity while Kistler [11] uses runtime path profiling.

Rabbah and Palem [12] introduce a concept of *neighbor affinity probability*, or NAP. It is the probability of a cache line containing successive data accesses. In their work, a structure whose NAP is smaller than a threshold, the structure is fully split.

Strout et al. [15] have studied data layout and loop iteration reordering opportunity for irregularly array access. They propose unified framework for array data layout, loop iteration reordering and sparse tilling. Their work is based on obtaining data access sequence by *runtime inspecting*. Although their work mainly focused on array, the idea of runtime data layout may also applied to structures.

Zhong et al. [16] defines a hierarchical model, called *reference affinity*, based on which, their further work [2] proposes *k-distance analysis* that are used to guide structure data-layout. *Reference affinity* is one way to measure affinity of different fields in the same instance. However as stated before, missing inter-instance affinity could misjudge some optimization opportunities.

Jeon et al. [17] propose a static analysis method to measure field affinity. They transform CFG into an automaton and merge all automatons throughout the program. Accesses to structure fields are abstracted into regular expression with the help of the automaton. All the repeated field access sequences in the regular expression are statically assigned repeating counts and fields affinities are calculated accordingly. If the affinity between two fields is larger than the sum of the self-affinity of the two fields, these two fields are merged into one. The merging process repeats until no fields fulfill the requirements and the merging result gives a data layout strategy.

Finally, Zhao et al. [2] extensively studied different data-layout strategies. They find maximal reshaping achieves the best or close-to-the-best performance in benchmarks they studied. Considering inter-instance affinity, we found that it is possible to find better layout plans than maximal reshaping, even in the benchmarks they studied.

7 Conclusions

In this paper, we showed that a structure data layout algorithm should take into account both intra-instance affinity and inter-instance affinity. We propose an algorithm that utilizes the two affinities and achieves better results than maximal reshaping, which is known to be close to the best for some programs.

In addition, we propose lightweight methods to estimate intra-instance affinity and inter-instance affinity efficiently. Together with execution frequency feedback, we use compile-time analysis to measure intra-instance affinity, and performance counters to measure inter-instance affinity. The total overhead introduced by our approach is about 4X the program execution time, and the storage requirement is very small and can be ignored. The overhead is much lower than that of the existing memory trace analysis approaches.

References

1 Zhong Y, Orlovich M, Shen X, et al. Array regrouping and structure splitting using whole-program reference affinity. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, 2004. 255–266

2 Zhao P, Cui S, Gao Y, et al. Forma: A framework for safe automatic array reshaping. ACM Trans Progr Lang Syst, 2007, 30: article 2

3 Petrank E, Rawitz D. The hardness of cache conscious data placement. In: Proceedings of the 29th ACM SIGPLANSIGACT Symposium on Principles of Programming Languages, 2002. 101–112

4 Open64 compiler. http://www.open64.net

5 Hundt R, Mannarswamy S, Chakrabarti D R. Practical structure layout optimization and advice. In: Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2006), 2006. 233–244

6 SPEC CPU 2000 Benchmark Suite. http://www.spec.org

7 SPEC CPU 2006 Benchmark Suite. http://www.spec.org

8 Rogers A, Carlisle M C, Reppy J H, et al. Supporting dynamic data structures on distributed-memory machines. ACM Trans Progr Lang Syst, 1995, 17: 233–263

9 Truong D N, Bodin F, Seznec A. Improving cache behavior of dynamically allocated data structures. In: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, 1998. 322–329

10 Chilimbi T M, Hill M D, Larus J R. Cache-conscious structure layout. In: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, Atlanta, Georgia, USA, 1999. 1–12

11 Kistler T, Franz M. Automated data-member layout of heap objects to improve memory-hierarchy performance. ACM Trans Progr Lang Syst, 2000, 22: 490–505

12 Rabbah R M, Palem K V. Data remapping for design space optimization of embedded memory systems. ACM Trans Embed Comput Syst, 2003, 2: 186–218

13 Curial S, Zhao P, Amaral J N, et al. Mpads: memory-pooling-assisted data splitting. In: Proceedings of the 7th International Symposium on Memory Management, Tucson, Arizona, USA, 2008. 101–110

14 Chilimbi T M, Davidson B, Larus J R. Cache-conscious structure definition. In: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, Atlanta, Georgia, USA, 1999. 13–24

15 Strout M M, Carter L, Ferrante J. Compile-time composition of run-time data and iteration reorderings. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, 2003. 91–102

16 Zhong Y, Shen X, Ding C. A hierarchical model of reference affinity. In: Languages and Compilers for Parallel Computing, 16th International Workshop, LCPC 2003, College Station, Texas, USA, 2003. 48–63

17 Jeon J, Shin K, Han H. Layout transformations for heap objects using static access patterns. In: Compiler Construction, 16th International Conference, March 26-30, 2007, Proceedings, Braga, Portugal, 2007. 187–201