Considerations on Dynamically Allocated Data Structure Layout Optimization

Dan Truong

Dan.Truong@enssat.fr	dtruong@irisa.fr
ENSSAT-LASTI	IRISA
6 rue de Kerampont	Campus de Beaulieu
22305 Lannion	35042 Rennes
FRANCE	FRANCE

Abstract

In a previous paper, we proposed two data layout optimizations for dynamically allocated heterogenous data structures, field reorganization and instance interleaving. We proposed that tools for these optimizations be provided within a performance tuning toolbox to help programmers fine-tune their applications.

In this communication, we investigate the possibility of automating these optimizations. We show that though we perform the optimization process manually, it is systematic and therefore likely to be automatable at least to some degree. Furthermore, some data-layout constraints related to these optimisations can be removed when the optimization process is automated.

Programmers may be reluctant to modify their programs to improve their performance, so automating such optimizations is a necessary step towards broad acceptance of such optimization techniques.

1. Introduction

In a previous paper [9], we proposed two data layout optimizations for dynamically allocated data structures, field reorganization and instance interleaving. We showed that by combining them we were able to improve the performance of memory bound applications which make intensive use of dynamically allocated data structures. We investigate in this communication the potential for automating such optimizations.

We first recall the two layout optimizations in section 2. The impact on program performance is shown separately for each optimization, on three different machines. Results show that combining the two optimizations provides more stable speedups. Optimized code samples are also given to shown that code transformations are minimal.

In section 3, the profile-based hand optimization steps are shown. The process is applied systematically, and therefore could be automated. We describe data layout optimization difficulties which limit the hand optimizations that can be performed, and indicate that most difficulties are removed if the optimization is automated.

We conclude in section 4.

The *Bench* benchmark follows three steps:

- It allocates and initializes a chained list of 512K instances of a structure (64K for the DEC machine).
- It verifies the integrity of data stored in the list.
- It performs a series of 512K searches in the list (64K for DEC).

The structure must hold at least three fields. The first two are used to search the list:

- The pointer to the next instance,
- The search key,
- One or more data elements (or pointers to data elements)

The data structure size can be chosen among 3 sizes:

- A small structure of 3 fields (12 bytes), smaller than the cache line size of the SGI machine.
- A medium structure of 8 fields (32 bytes), of the same size as the cache line of the SGI machine.
- A large structure of 24 fields (96 bytes), three times larger than the cache line size of the SGI machine.

Figure 1. The synthetic benchmark *bench*

Library	Layout	Bench-3 512 512 1	Bench-8 512 512 1	Bench-24 512 512 1	Ft 20000 30000	Health 6 500 1
SUN Ultra Entreprise / Ultra4 UltraSparc-II						
BSD	Original	22.5	41.4	55.0	15.4	29.1
	Reorganized	22.6	28.1	33.8	-	29.0
	Gain	**-0%**	**32%**	**39%**	-	**0%**
GNU	Original	30.9	39.7	47.1	14.6	25.2
	Reorganized	25.3	26.8	30.5	-	24.0
	Gain	**18%**	**32%**	**35%**	-	**5%**
Ialloc	Original	19.2	37.9	45.4	13.8	25.6
	Reorganized	19.2	26.1	28.9	-	25.4
	Gain	**0%**	**31%**	**36%**	-	**1%**
	Interleaved	18.1	16.4	16.4	6.6	25.1
	Gain	**6%**	**57%**	**64%**	**52%**	**2%**
	Reorg+Intl	16.3	16.4	16.4	(6.6)	19.2
	Gain	**15%**	**57%**	**64%**	**(52%)**	**25%**
SGI Onyx / IP27 R10000						
SGI	Original	21.70	29.89	33.78	36.80	39.09
	Reorganized	21.65	27.14	31.68	-	39.76
	Gain	**0%**	**9%**	**6%**	-	**-0%**
Ialloc	Original	15.22	26.84	34.33	19.86	30.57
	Reorganized	15.26	26.54	32.06	-	30.51
	Gain	**-0%**	**1%**	**7%**	-	**0%**
	Interleaved	14.70	11.97	14.81	7.53	21.50
	Gain	**32%**	**55%**	**57%**	**62%**	**30%**
	Reorg+Intl	12.03	12.02	12.03	(7.53)	17.57
	Gain	**45%**	**55%**	**65%**	**(62%)**	**43%**
DEC Station 3000-300 / Alpha 21064						
DEC	Original	17.3	22.4	38.9	11.7	127.4
	Reorganized	17.5	17.7	23.9	-	121.2
	Gain	**-0%**	**21%**	**39%**	-	**5%**
Ialloc	Original	12.4	19.5	34.1	11.7	113.4
	Reorganized	12.6	19.7	24.3	-	110.3
	Gain	**-2%**	**-1%**	**29%**	-	**3%**
	Interleaved	13.3	13.2	13.3	11.8	108.0
	Gain	**-7%**	**32%**	**61%**	**-0%**	**5%**
	Reorg+Intl	11.5	11.5	11.3	(11.7)	85.4
	Gain	**7%**	**41%**	**67%**	**(0%)**	**25%**

Table 1. Field reorganization and instance interleaving reduce program user time separately but combining them is best.

2. Data layout optimization for dynamically allocated data structures

Most of the cache misses are usually caused by accesses to dynamically allocated data structures or arrays [6, 5], because these data types are used to store most of the data manipulated by the programs. Many layout techniques have already been proposed to optimize arrays and have been shown to be very efficient [2].

Therefore, our data-layout optimizations focus on dynamically allocated heterogenous data structures. We proposed in [9] two profile-based data layout techniques, field reorganization and instance interleaving, to improve the spatial and temporal locality of dynamically allocated data structures.

As opposed to hardware solutions like cache bypassing [10] or prefetching [4], software layout optimizations can improve spatial locality within memory blocks by regroupping frequently used data elements together. Therefore after layout optimization, a cache line is more likely to hold data elements with high spatial locality, effectively reducing capacity misses as well as conflict misses.

We provide the execution time of three benchmarks on three different machines in figure 1. Results are given for the original and optimized layouts for standard *malloc* libraries and our customized *ialloc* library. Speedups are given in precent relative to the original layout using the same library. For example, the fully-optimized *Bench-24* is 64% faster than the original version on *Sun* using *ialloc* (16.4s versus 45.4s). Note that if we compare to the *BSD malloc* (55s), the speedup becomes 70%, because *ialloc* manages memory differently than *BSD malloc*. The *DEC* machine we used doesn't have much memory, so some tests have been done with a smaller data set, as specified in the text. The test machines used are described in table 2. The synthetic benchmark *Bench* described in figure 1 is built to show the impact of the optimizations on programs which spend most of their time referencing dynamically allocated instances of data structure. *Bench* spends a negligible amount of time allocating or testing integrity of the instances.

Field reorganization It is performed in C by regroupping the most frequently used fields of a data structure together:

```
BEFORE                              AFTER

typedef struct _TheStruct {         typedef struct _TheStruct {
  struct _TheStruct * Next;           struct _TheStruct * Next;
  int Data2;                          int Comp;  /**/
  int Data3;                          int Data;
  int Data4;                          int Data2;
  int Comp;  /**/                     int Data3;
  int Data5;                          int Data4;
  int Data6;                          int Data5;
  int Data;                           int Data6
} TheStruct;                        } TheStruct;
Pt = (TheStruct *)                  Pt = (TheStruct *)
  malloc(sizeof(TheStruct));          malloc(sizeof(TheStruct));
```

The code sample is taken from the *Bench-8* program. Profiling (discussed in section 3) indicates that fields *Next* and *Comp* are the most frequently used and should be regroupped. Only the data structure declaration is modified in the program.

This optimization can reduce the number of cache lines loaded when the program uses data structures larger than the cache line size. Programmers are rarely aware that the the order of the fields in a structure can have an impact on performance.

Table 1 shows the performance gains obtained by field reorganization for different allocation libraries.

Field reorganization alone is efficient for data structures larger than the line size, but is not when the structures are smaller. With larger structures, reorganization can reduce the number of memory blocks that must be loaded in the cache when an instance is referenced (speedups range from 6% to 39% for *Bench-24*). With smaller structures, fields are just reorganized within a memory block, so the performance impact is smaller (speedups from -2% to 18% for *Bench-3*). The *Onyx* isn't very sensitive to field reorganization, probably because its L2 data cache uses lines larger than the *Bench-24* structure (128 bytes), while the *Sun* machine is more sensitive because it uses a sectored L1 data cache with 16 bytes per sectors.

Since programs often use small data structures [11], field reorganization alone may not be sufficient to speed-up programs.

Instance interleaving It is done with the help of the *ialloc* library, a special allocation library that we developped to support instance interleaving [9]. Padding arrays are inserted between each field in the structure declaration and the instances of the structure are allocated with *ialloc/ifree*, instead of *malloc/free*. *Ialloc* reuses the padding space to allocate the fields of other instances, therefore effectively interleaving the fields of different instances.

The code transformations necessary can be minimal as shown below for the *Ft* benchmark [1]:

```
BEFORE                        AFTER

graph.h:                      graph.h:

                              #include "ialloc.h"
                              #define FREE0(pt) ifree(pt)
                              #define ALLOCATE0 ialloc(Vertices, CHUNKSIZE0, 0)
                              #define CHUNKSIZE0 sizeof(void*)

typedef struct _Vertices      typedef struct _Vertices
{                             {
  int              id;          int              id;
                                char Pad1[_IA_INTERLEAVE - sizeof(int)];
  Edges * edges;                Edges * edges;
                                char Pad2[_IA_INTERLEAVE - sizeof(void*)];
  struct _Vertices * next;      struct _Vertices * next;
                                char Pad3[_IA_INTERLEAVE - sizeof(void*)];

  /*                            /*
   * For the ft algorithm.       * For the ft algorithm.
   */                            */
  int              key;         int              key;
                                char Pad4[_IA_INTERLEAVE - sizeof(int)];
  Edges * chosenEdge;           Edges * chosenEdge;
```

```
                               char Pad5[_IA_INTERLEAVE - sizeof(void*)];
} Vertices;                  } Vertices;

graph.c:                     graph.c:

vertex = (Vertices *)
    malloc(sizeof(Vertices));   vertex = ALLOCATE0;
```

The code given shows all the modifications done to the *Ft* program to interleave the data structure holding the most frequently used data. The algorithm seaches a chained list, using only the *next* field so interleaving is the only optimization we applied to it[1]. Interleaving speeds up *Ft* by up to 62%[2]. *Ft* has a total of 2153 lines of code and spans over 11 source files.

_IA_INTERLEAVE is the interleave factor, a constant provided by the *ialloc* library. It is slightly smaller than the cache size. Therefore, the frequently used fields interleaved can fill the whole cache without interfering to maximize temporal locality, but when a single instance is accessed, the different memory blocks holding its fields won't interfere together because they map to distinct cache sets.

This optimization increases spatial locality between identical fields of different instances. It is most efficient when a few fields of the structure are referenced frequently. Results in table 1, show that interleaving can even improve the performance of programs using small structures (up to 32% for *Bench-3*) as well as of those using large structures (up to 64% for *Bench-24*). There is a degradation of performance on the DEC machine of 7%. Since two out of the three fields are used frequently in *Bench-3*, interferences may degrade performance.

Combined optimizations We combine these two techniques to optimize programs and reduce the risks of cache interferences. Fields frequently used together are first regroupped into a chunk. The other fields of the structure are groupped into more chunks of the same size (or smaller) as the first chunk. Padding arrays are inserted between the chunks and the structure is allocated with the *ialloc* library to be interleaved, as shown for the *Health* benchmark [7, 3, 4]:

```
BEFORE

health.h:

struct Patient {
  int                   hosps_visited;
  int                   time;
  int                   time_left;
  struct Village        *home_village; };

struct List {
  struct Patient        *patient;
  struct List           *back;
  struct List           *forward; };

health.c:

...

list.c:

list = (struct List *)mymalloc(sizeof(struct List));
```

```
AFTER

health.h:

#include "ialloc.h"
#define FREE1(pt) ifree(pt)
#define ALLOCATE1(i,j) ialloc(struct Patient, CHUNKSIZE1 ,1)
#define CHUNKSIZE1  (sizeof(void *))

struct Patient {
  int                   time;
  unsigned char Dummy0[_IA_INTERLEAVE - sizeof(int)];
  int                   hosps_visited;
  unsigned char Dummy1[_IA_INTERLEAVE - sizeof(int)];
  int                   time_left;
  unsigned char Dummy2[_IA_INTERLEAVE - sizeof(int)];
  struct Village        *home_village;
  unsigned char Dummy3[_IA_INTERLEAVE - sizeof(void *)];
 };

#define ALLOCATE2 ialloc(struct List, CHUNKSIZE2 ,2)
#define CHUNKSIZE2  (2 * sizeof(void *))

struct List {
  struct Patient        *patient;
  struct List           *forward;
  unsigned char Dummy0[_IA_INTERLEAVE - 2*sizeof(void *)];
  struct List           *back;
  unsigned char Dummy1[_IA_INTERLEAVE - sizeof(void *)];
}


health.c:

new->hosp->assess = ALLOCATE2;
new->hosp->waiting = ALLOCATE2;
new->hosp->inside = ALLOCATE2;
new->hosp->up = ALLOCATE2;
new->returned = ALLOCATE2;
patient = ALLOCATE1;

list.c:

l = (struct List *)ALLOCATE2;
```

The structure *Patient* is interleaved field by field, but *List* is split into two chunks of two words. Note how the padding array is declared for the *List* structure. The last chunk holds only one field, but the padding array declaration also straightforward. However since the last chunk is half-filled, interleaving wastes 4 bytes per instance in that case.

The combied optimizations provide speedups for all data structure sizes, as shown in rows "Intl+Reorg" of table 1. Notice that the execution times are now the same for the 3 *Bench* programs. The size of the data structure has been factored out and does not impact on cache performance any more.

2.1. Conclusion

The two data layout optimizations are both effective to improve dynamically allocated data structure spatial locality. However, combining them is necessary to reduce the risks of cache interferences and to maximize speedups. With proper data layout optimization, rarely used fields are no longer loaded in the cache with frequently used ones, and do not impair cache performance any more.

The data layout optimization proposed are applied on programs using dynamic execution informations gathered by profiling tools. In the next section, we discuss on the possibilities of automating the profiling and optimization

[1] That is why there is no entry for field reorganization in table 1 for *Ft*.
[2] We ran "ft 500 600" for the DEC machine.

Machine	Vendor	CPU	Frequency	Mode	L1 Dcache	Assoc	Line	L2	DRAM
Onyx	SGI	R10000	195MHz	32bit	32KB	2-way	32B	1MB	512MB
Ultra Entreprise	SUN	Ultrasparc2	296MHz	32bit	16KB	direct	2×16B	2MB	1.1GB
DecStation 3000-300	DEC	21064	150MHz	64bit	8KB	direct	32B	256KB	59MB

Table 2. Hardware description of the machines used.

process.

3. Profile-based data layout optimization

The layout optimizations we propose necessitate profiling and code transformations which are done by hand. This is a restriction to the usability of the method we propose.

The *ialloc* library was first developped to hide most of the complexity of memory management for interleaved data structures. However, data declarations must also be modified, so the code transformations necessary for optimization cannot be fully hidden to the programmer. Therefore, many factors could dissuade the programmers from using these layout optimizations:

- The programmer must spend time to profile and optimize the code by hand.
- Since the code is transformed to fool the compiler into doing instance interleaving, it becomes less readable and thus more difficult to maintain for software engeneering.
- Some programming styles and the usage of non-scalar data types in the optimized structures can increase the difficulty of hand optimizations.

We have shown in [9] that a profile based optimization was likely to provide consistant speedups for different input sets, giving strong incentives to the programmers for using such optimizations. However, these incentives may not be sufficient when software engeneering constraints are taken into account. Ideally, this optimization should be automatic and transparent to the programmer. There are 3 steps in our optimization process, profiling, field reorganization and instance interleaving, which we expect can be automated to some degree.

3.1. Profiling

It is done using readily available tools, which are already used for performance tuning.

We used SGI's Speedshop package, including Perfex and Ssrun. Ssrun is built on top of Pixie [8] and uses hardware counters of the R10000 CPU to count among other things the number of L1 data cache misses caused by program C-source code lines.

Often only a few references in the program are responsible for the majority of the cache misses [10]. Profile information helps to focus the optimizations onto a few hot spots in the program.

For example, the profile obtained for the *Ft* benchmark indicates that misses come from a single source code line (line 170 of *PickVertex()* in *graph.c*):

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Mon Aug 24 12:14:32 1998
   prof -h FT.GMalloc.Orig.dc_hwc.m1705481
                     dc_hwc: Experiment name
             hwc,25,2053:cu: Marching orders
...
-------------------------------------------------------------------------
Summary of R10K perf. counter overflow PC sampling data (dc_hwc)--
   Primary cache D misses (25): Counter name (number)
...
-------------------------------------------------------------------------
Line list, in descending order by counts
-------------------------------------------------------------------------
        counts     %   cum.%   samples  function (dso: file, line)

     298555472   99.0   99.0    145424  PickVertex (FT.GMalloc.Orig: graph.c, 170)
       1022394    0.3   99.3       498  DeleteMin (FT.GMalloc.Orig: Fheap.c, 255)
        180664    0.1   99.4        88  MST (FT.GMalloc.Orig: ft.c, 171)
        149869    0.0   99.4        73  _malloc (libc.so.1: malloc.c, 183)
...
```

With this information, we know which data structure must be optimized by reading the source code:

```
    graph.h

132 #define NEXT_VERTEX(V)  ((*(V)).next)

    graph.c

163 Vertices *
164 PickVertex(Vertices * graph, int whichVertex)
165 {
166  int       i;
167
168  for(i = 0; i < whichVertex; i++)
169  {
170    graph = NEXT_VERTEX(graph);
171  }
```

For *Ft*, line 170 of *graph.c* references the *next* field of the *Vertices* structure. The optimized code shown in the previous section in the paragraph *Instance interleaving* just interleaves the *next* fields of the instances of the *Vertices* structure.

This type of analysis could be added in modern compilers, since current compilers like SGI's C MipsPro compiler already take advantage of profile feedback to fine-tune software pipelining and instruction layout.

The only drawbacks left is that the programmer must test the program with a significant input set and that he must compile the program twice.

3.2. Field reorganization

All the most frequently used fields must be detected using profile information and code analysis, to be regroupped together.

The frequently used fields which generate cache misses are detected directly by the profile, but frequently used fields do not necessarily generate cache misses if they are systematically prefetched in the cache lines while the other fields are referenced.

When a source line causing misses is found, the whole program block[3] (loop nest or procedure) must be considered to find all the fields of the structure referenced along with the field causing misses. These fields may not cause misses if they are prefetched by the cache miss. If they are not selected for optimization however, they are likely to cause misses because they won't be interleaved with the other frequently used fields.

```
    health.c

219 void check_patients_waiting(struct Village *village, struct List *list)
220 {
221   int                    i, t;
222   struct List            *l;
223   struct Patient         *p;
224
225   /*MIGRPH();*/
226   while (list != NULL) {
227     i = village->hosp.free_personnel;
228     p = list->patient;                    /**/
229     if (i > 0) {
230       t = village->hosp.free_personnel;
231       village->hosp.free_personnel = t-1;
232       p->time_left = 3;
233       NOTEST();
234       t = p->time;
235       p->time = t + 3;
236       RETEST();
237       l = &(village->hosp.waiting);
238       removeList(l, p);
239       l = &(village->hosp.assess);
240       addList(l, p); }
241     else {
242       t = p->time;                        /**/
243       p->time = t + 1; }                  /**/
244     list = list->forward; }               /**/
245 }
```

For *Health*, profile information indicates that 75% of the cache misses are caused by lines 228 and 242 in the file *health.c*. Code executed with these lines include the *while* loop body. The *if* condition isn't executed with the body, but the *else* is. Extra profile information can confirm which code sections are most executed, like a count of the number of times a line is run. Here we see that two structures have many instances referenced, but few fields accessed, including *Patient.time*, *List.patient* and *List.forward*. Those fields have been optimized in the previous section. Note that line 244 does not cause many cache misses (4%), because the *forward* field is prefetched by the cache line along with *patient* field. Line 227 causes only 0.5% misses because there are few instances of *Village*. After optimization, lines 228 and 242 still cause most of the misses, but misses for lines 243 and 244 drop to zero and the misses are reduced by 73%.

Using profile information and simple code analysis, a compiler obtains a sufficient amount of information to detect all the frequently used fields just like we do by hand. Therefore, field reorganization is likely to be automatable too.

[3]All the source lines which are likely to be executed systematically along with the source line causing misses must be considered.

3.3. Code/Data transformations for instance interleaving

The difficulty of the hand-optimization process lies in the code and data transformations that must be done to perform instance interleaving. The results of section 2 indicate that interleaving can provide important performance gains, so this optimization step should not be left out, even though it is more complex to apply.

3.3.1 Difficulties with data types

Transformations are straightforward for data structures holding only scalar data types. Padding arrays are just added to the data structure declaration and calls to the *malloc* library are redirected to the *ialloc* library, as is done in code samples of section 2.

When non-scalar data types are encountered, code transformations can become more difficult to perform, because large data types may prevent efficient splitting of a data structure into chunks. When the elements are larger than the chunk size chosen, a solution is to replace them by a pointer in the structure and to allocate them separately outside the structure. However, since this method inserts an indirection (reference through a pointer), it may necessitate to change all the references to the element, depending on the type of the element.

Arrays and strings When a structure holds tables or strings, these must be allocated separately. The declaration is changed into a pointer, and the table or string is allocated at the same time as the structure instance, as shown below.

```
BEFORE                        AFTER

typedef struct _X {           typedef struct _X {
  int scalar;                   int scalar;
                                char Pad1[IA_INTERLEAVE - sizeof(int)];
  char Tab[256];                char * Tab; /* Convertion */
                                char Pad2[IA_INTERLEAVE - sizeof(void*)];
} X                           } X

X x;                          X x;
x = (X *) malloc(sizeof(X)); x = ialloc(X,sizeof(int),0);
                              x->Tab = (char*) malloc(256*sizeof(char));
x->Tab[0] = '\0';             x->Tab[0] = '\0'; /* references are unchanged */
```

The code transformations to perform are still minor, though the data organization changes in memory. An extra malloc must be inserted, but references to the structure are unchanged.

Data structures When a sub-structure is used within a structure, using the same approach as above is difficult by hand, because all references to the sub-structure must be modified. The last line of the example below shows that all references to field *A* of the sub-structure *SubStr* are modified when the indirection is inserted:

```
BEFORE                         AFTER

typedef struct _SubStr {       typedef struct _SubStr {
  int A;                         int A;
  int B;                         int B;
} SubStr                       } SubStr

typedef struct _X {            typedef struct _X {
  int scalar;                    int scalar;
                                 char Pad1[IA_INTERLEAVE - sizeof(int)];
  SubStr S;                      SubStr * S; /* Convertion */
                                 char Pad2[IA_INTERLEAVE - sizeof(void*)];
} X                            } X

X x;                           X x;
x = (X *) malloc(sizeof(X)); x = ialloc(X,sizeof(int),0);
                               x->S = (char*) malloc(sizeof(SubStr)];
x->S.A = 0;                    x->S->A = 0; /* All references are changed */
```

We modified the structure *Hosp* of the *Health* benchmark this way, to allow interleaving of the structure *List*, as shown in the previous chapter in paragraph *Combined optimizations*. In the code listing showed, extra allocation lines appear for all the pointers of the *Hosp* structure. These extra indirections generated do not reduce the program performance. Without transformations the user time is 39.81s, and with the convertions into pointers, 39.09s.

Other layout choices can be made for sub-structures. For example, if the sub-structure is never used separately (i.e. no pointers to it are used), the sub-structure can be colapsed to be replaced by its fields as shown below.

```
BEFORE                         AFTER

typedef struct _X {            typedef struct _X {
  int scalar;                    int scalar;
                                 char Pad1[IA_INTERLEAVE - sizeof(int)];
  SubStr S;                      int S_A;
                                 char Pad2[IA_INTERLEAVE - sizeof(void*)];
                                 int S_B;
                                 char Pad2[IA_INTERLEAVE - sizeof(void*)];
} X                            } X

X x;                           X x;
x = (X *) malloc(sizeof(X)); x = ialloc(X,sizeof(int),0);
x->s.A = 0;                    x->S_A = 0; /* All references are changed */
```

This solution removes the indirection but all references to the structure still have to be modified.

Pointer arithmetic and casting A major difficulty for data layout optimization comes from pointer arithmetic and casting.

```
X x;
((int *)x)[2] = 0;
```

A programmer has the right to access the fields of an instance of a data structure not just by referencing explicitely the name of the field, but also by computing its address in the structure from the base address of the instance. Catching and correcting such address computations is difficult by hand and may also be a problem for automatic optimization. However such a method is not frequent, and we haven't met this problem in the benchmarks tested.

A more frequent problem is the casting of an instance of a structure to another type. This can be used for instance when the programmer manages memory allocation himself (as in the *Health* benchmark). The allocation routine returns an array of a given size. The array returned is then casted into the correct structure.

```
void * Alloc(int Size) {
  char * Pt;
  Pt = malloc(Size);
  if (Pt) return Pt;
  exit(1);
}

x = (X *) Alloc(sizeof(X));
```

Structure casting can also be used in C to let a procedure manipulate different structures without distinction, like children of a class in C++.

Hand optimization becomes difficult when the structures to optimize are casted. This is also likely be a problem for automatic optimization. To elude this difficulty at first, data layout optimizations (including field reorganization) can be canceled if casting or unsupported pointer arithmetic is detected when the code is parsed.

3.3.2 Instance organization in the heap

When the instances of the data structure to optimize are allocated separately one by one and managed as recursive data structures (i.e. in chained lists or trees), they can be interleaved by hand. When the instances are embedded in an array, interleaving can hardly be done by hand.

Array of structures If the interleaved instances are stored within an array, the programmer must fool the compiler into doing the proper pointer arithmetic. When the i-th element in a table is referenced, the compiler takes the base address of the table and increments it autmatically by i-times the size of the structure. However, with an interleaved data structure, the address of the i-th element becomes *i*-times the size of a chunk. Therefore with hand optimizations, when an element is referenced the table must be casted prior to address computation. A fake data structure the size of a chunk must be made for the cast. A correct way to reference a table would be:

```
BEFORE                         AFTER

Tab[i].scalar                  (*((Struct *) (((Fake *) Tab) + i ))).scalar
```

Modifying all the references to the table this way is too complex to be done by hand, but can be done automatically. Furthermore, if the optimization is done by the compiler, these code transformations become unecessary. The compiler just needs to know that for interleaved data structures, pointer arithmetic is done using the chunk size and not the structure size.

Therefore, with automatic data layout optimization, interleaving can be extended to arrays of structures, provided that the allocation library provides support for the allocation of arrays.

3.4. Locality gains

The data layout increases spatial locality. The ouput generated by SGI's *Perfex* tool demonstrates this by indicating that cache line reuse is increased and memory bandwidth used is reduced. It uses hardware counters to count the number of CPU events.

For *Ft* the time lost in L1 and L2 is preponderant with the original layout (26s) but is decreased signigficantly with interleaving (1.5s), as the *Perfex* output shows in figure 2. *Perfex* computes that the L1 D-cache line reuse increases from 0.5 to 14.2, that bandwidth between L1 and L2 data caches is reduced from 266MB/s to 136MB/s and that the hit ratio improves from 34% to 93%.

Improved cache line reuse reduces the average memory latency while reducing at the same time memory bandwidth consumed. On the other hand, hardware optimizations like cache bypassing [10] or prefetching improve average memory latency but may require extra memory bandwidth, since they do not change the program's spatial locality.

4. Conclusion

We proposed in a previous paper two profile-based data layout optimizations, field reorganization and instance interleaving. In many cases, the optimizations can be performed by hand by modifying just the data structure declaration and the calls to *malloc* and *free* for that structure, and provide significant speedups by improving cache spatial locality. Though often straigtforward to perform, manual optimization is a limiting factor which reduces the scope of application of the optimization.

In this paper, we explore the potential for automatization of the optimization process. We first expose the profile based manual optimization method we used and the restrictions encountered with hand optimizations. The hand optimization process uses a systematic approach, therefore it is likely to be automatable. Furthermore, automatization of the optimizations can increase the scope of applicability. It is no longer necessary to minimize the extent of source code transformations. By automating the optimizations, complex structure organizations can also be optimized, including structures holding complex data types, or structures allocated within arrays. Furthermore, if the optimizations are inserted into a profiling compiler, the constraints comming from the grammar of the language are removed.

With automatic optimization, correctness of the transformed code must be guaranteed, therefore when the layout optimization isn't safe, it should be disabled. The C language is not well suited for such an optimization, because it is very permissive to the programmers, compared to modern languages like Java which do not allow the programmer to manipulate pointers and hides memory management. Therefore, it may be difficult to perform thorough pointer aliasing analysis. Such an analysis is necessary if pointer arithmetic or casting is performed on the structures to optimize. The most frequent source of structure casting we have found is programs which handle memory management with custom routines to improve performance. This becomes unecessary when interleaving is performed and conflicts with the optimizations we propose. When the code is too difficult to analyse to guarantee the correctness of the code transformations, the optimizations can be canceled, and it is up to the programmer to simplify his program to remove all ambiguities. For most of the other programs, this limitation should not prevent the optimization process.

Instance interleaving and field reorganization can provide significant speedups to non-scientific applications. However, these optimizations have yet to be automated. This paper shows that hand optimizations follows a systematic approach and therefore could be automated.

References

[1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, http://www.cs.wisc.edu/ austin/austin.html, June 1994.

[2] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, Dec. 1994.

[3] M. C. Carlisle and A. Rogers. Software caching and computation migration in olden. Technical Report TR-483-95, Princeton University, 1995.

[4] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *7th symposium on Architectural Support for Programming Languages and Operating Systems*, 1996.

[5] S. Mehrotua and L. Harrison. Examination of a memory access classification scheme for pointer-intensive and numeri-

```
PERFEX OUTPUT ORIGINAL/MALLOC CODE
                                                    Event definitions for cpu version 2.x
                                                                                    Typical      Minimum      Maximum
   Event Counter Name                                            Counter Value   Time (sec)   Time (sec)   Time (sec)
=====================================================================================================================
 0 Cycles......................................................    6631234576    36.840192    36.840192    36.840192
25 Primary data cache misses...................................     305132736    15.273589     4.780413    15.273589
26 Secondary data cache misses.................................      25644512    10.756448     7.032295    11.967439
 6 Decoded branches............................................     553872704     3.077071     3.077071     3.077071
...

Statistics
=========================================================================================
Graduated instructions/cycle................................................     0.267539
Graduated loads & stores/cycle..............................................     0.069894
Data mispredict/Data scache hits............................................     0.094676
Instruction mispredict/Instruction scache hits..............................     0.113392
L1 Cache Line Reuse.........................................................     0.518950
L2 Cache Line Reuse.........................................................    10.898559
L1 Data Cache Hit Rate......................................................     0.341651
L2 Data Cache Hit Rate......................................................     0.915956
Time accessing memory/Total time............................................     0.778169
L1--L2 bandwidth used (MB/s, average per process)...........................   266.387965
Memory bandwidth used (MB/s, average per process)...........................    90.212707
MFLOPS (average per process)................................................     0.000000
```

```
PERFEX OUTPUT INTERLEAVED/IALLOC CODE

   Event Counter Name                                            Counter Value   Time (sec)   Time (sec)   Time (sec)
=====================================================================================================================
 0 Cycles......................................................    1333050336     7.405835     7.405835     7.405835
25 Primary data cache misses...................................      30491840     1.526286     0.477705     1.526286
26 Secondary data cache misses.................................        193904     0.081332     0.053173     0.090489
 6 Decoded branches............................................     561260256     3.118113     3.118113     3.118113
...
Statistics
=========================================================================================
Graduated instructions/cycle................................................     1.310886
Graduated loads & stores/cycle..............................................     0.348023
Data mispredict/Data scache hits............................................     0.102796
Instruction mispredict/Instruction scache hits..............................     0.053903
L1 Cache Line Reuse.........................................................    14.214965
L2 Cache Line Reuse.........................................................   156.252249
L1 Data Cache Hit Rate......................................................     0.934275
L2 Data Cache Hit Rate......................................................     0.993641
Time accessing memory/Total time............................................     0.566760
L1--L2 bandwidth used (MB/s, average per process)...........................   135.693251
Memory bandwidth used (MB/s, average per process)...........................     4.514563
MFLOPS (average per process)................................................     0.000000
```

Figure 2. Optimization statistic of the *Ft* benchmark.

cal programs. In *International Conference on supercomputing*, page 133, 1996.

[6] T. Ozawa, Y. Kimura, and S. Nishizaki. Cache miss heuristics and preloading techniques for general-purpose programs. *28th Annual International Symposium on Microarchitecture*, 1995.

[7] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2), Mar. 1995.

[8] M. D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford University, ftp://steward.harvard.edu/users/smith, Nov. 1991.

[9] D. N. Truong, F. Bodin, and A. Seznec. Improving locality of dynamically allocated data structures. In *PACT'98*, Oct. 1998.

[10] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. *28th Annual International Symposium on Microarchitecture*, 1995.

[11] B. Zorn and D. Grunwald. Empirical measurment of six allocation-intensive c programs. *ACM Sigplan Notices*, 27(12):71–80, Dec. 1992.