Improving cache behavior of dynamically allocated data structures

Dan N. Truong
dtruong@irisa.fr

François Bodin
bodin@irisa.fr

André Seznec
seznec@irisa.fr

IRISA - INRIA
Campus de Beaulieu
35042 Rennes, CEDEX
France

Abstract

Poor data layout in memory may generate weak data locality and poor performance. Code transformations such as loop blocking or interchanging and array padding have addressed this issue for scientific applications. However many generalist applications do not use data arrays, but dynamically allocated heterogeneous data structures.

In this paper, we explore two data layout techniques for dynamically allocated data structures: field reorganization, and instance interleaving. The application of these techniques may be guided by program profiling. This allows significant cache behavior improvements on some applications.

To support instance interleaving, we developed a specific memory allocation library called ialloc. An ialloc-like library may be of great help in a toolbox for performance tuning of general-purpose applications.

1. Introduction

Recent studies have analyzed the efficiency of caches. If cache usage was optimal, only small cache sizes would be necessary [11], and the memory bandwidth necessary would be kept low [12]. However, in practice, caches are greatly under-exploited [4], and memory bandwidth will be a limiting factor for processor performance because of limited chip pin-out [3]. Meanwhile, all efforts to better exploit effective cache space and available memory bandwidth are worth to be explored. Changing the reference patterns to memory locations modifies cache behavior. These changes can either be done by reordering the memory accesses of the application through code transformations, or by changing the layout of referenced data in memory. These techniques have been widely explored for numerical scientific applications. Program developers can rely on efficient cache-conscious numeric libraries [6] or on loop transformations [2, 17] (code transformations) or array padding [1] (data layout techniques) to improve locality.

However generalist applications may also require high performance and can suffer from poor cache behavior. In this paper, we focus on applications using heterogeneous data structures. Instances of structures are often allocated dynamically, either organized as self-referencing data structures (lists, B-trees, quad trees...), or as arrays of structures. Such applications will not benefit from the techniques developed for arrays to improve cache locality. Data addresses are not computed using loop indexes, so loop transformations cannot be used. Code transformations are unlikely to generate stride one memory accesses for instances of structures.

In this paper, we explore two possible data layout techniques to improve locality for heterogeneous data structures allocated dynamically.

- Field reorganization : fields of a data structure often referenced together are grouped together in the data structure declaration to fit in the same cache line.
- Instance interleaving : identical fields of different instances of a data structure often referenced together are grouped together dynamically. That way, rarely used fields are moved away from frequently used fields, and won't be loaded in the cache.

The C language does not allow direct implementation of the second data layout optimization. This leads us to develop a dedicated allocation library called *ialloc* to support the interleaving optimization. The usage of *ialloc* guided by profiling is shown to allow significant performance gain on some applications using extensively dynamically allocated data structures. The *ialloc* library might be a very valuable piece in a toolbox for performance tuning general applications.

The remainder of the paper is organized as follows. Section 2 presents in more details field reorganization and instance interleaving. Section 3 presents the dedicated memory allocation library *ialloc*. Section 4 shows our data layout experimentation using *ialloc*. We overview related works in section 5. Section 6 concludes this study.

2. Data layout

Ideally, data elements most often referenced together should lie into the same cache blocks [16, 21]. Furthermore, memory blocks frequently used together must map into distinct cache sets. To achieve this one would like to take advantage of properties of the programs and of the data structures to choose the data layout.

Locality properties can be found for dynamically allocated instances of data structures because the program manages instances anonymously using pointers. A data structure is generally made of a small number of fields (from two to a few dozens). Every instance holds these fields. Most instances are referenced through the same code portions. Therefore, the dominant reference patterns are usually the same for most instances of the structure. The dominating reference pattern is usually caused by a very small portion of the program [17]. A specific reference pattern only observed for a few instances of the structure can be ignored. It will not influence overall program performance. Therefore to improve the performance of a program, we must adapt the layout of the instances in memory to the dominating reference patterns.

In this section, we describe field reorganization and instance interleaving, two techniques that can be used to modify the layout of data structures in memory.

2.1. Field reorganization

In many applications, the most active parts of the code access only a few fields in each instance of a structure. These fields can be regrouped together to fit into the fewest number of memory blocks.

Figure 1 illustrates such a reorganization in C language to regroup fields A and C. The declaration of the data structure is just changed, modifying the memory layout of the fields of the instances. We call this optimization field reorganization.

Field reorganization can have a significant impact on cache performance if the structure spans over many cache blocks. When an instance is referenced, fewer memory blocks are loaded in the cache. This not only takes advantage of cache line prefetching, it also reduces cache pollution. Repeated over many instances, this optimization alone can provide visible speedups.

Unfortunately, programs usually don't use such large data structures. Therefore field reorganization alone becomes useless, because reorganizing the fields does not change the cache line reference pattern. To solve this, field reorganization can be combined with instance interleaving.

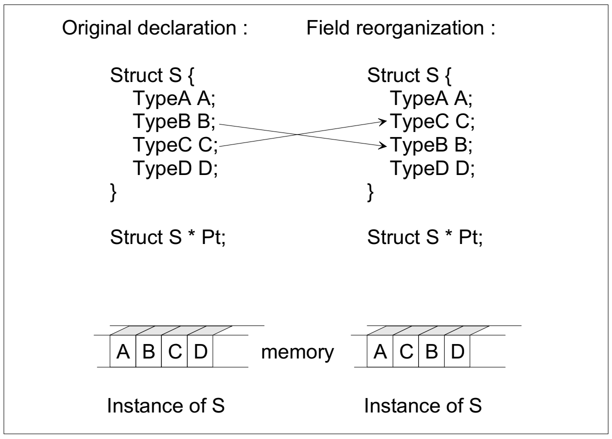

Figure 1. Field reorganization is done by changing the declaration order of the fields of the structure.

2.2. Instance interleaving

The dominating reference pattern of a program often accesses only a few fields in each instance, not enough to fill a cache line. Instead of filling a memory block with rarely used fields of the instance, one may want to fill it with frequently used fields of other instances. Therefore, these fields must be laid out contiguously in memory.

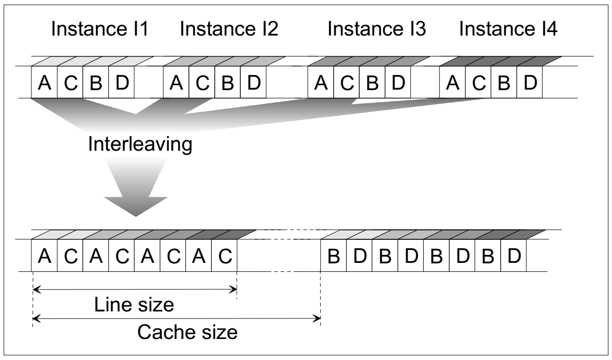

Figure 2. Interleaving of identical fields of many instances of a structure.

To achieve this, we propose to interleave the instances of the data structure as illustrated in Figure 2. We assume that the fields A and C are the most frequently used fields of the structure. We group them in the first chunk, while we group B and D in the second chunk. The chunks (A,C) of a few instances are grouped together to fill-up a cache block.

Since they are more likely to be referenced than the second chunks (B,D) , cache line reuse is likely to improve.

We actually interleave many more instances. We fill a whole segment of memory at least as large as the cache size with chunks (A,C). Since these chunks are all likely to be used, they should not map to the same cache sets. Organizing them contiguously in memory reduces the risk of cache interference to eliminate conflict misses.

It has been shown that instances allocated at the same time tend to be used at the same time [10]. By interleaving together the frequently used chunks of instances allocated consecutively, we prevent interferences between these instances and we increase the chances of taking advantage of cache line prefetching.

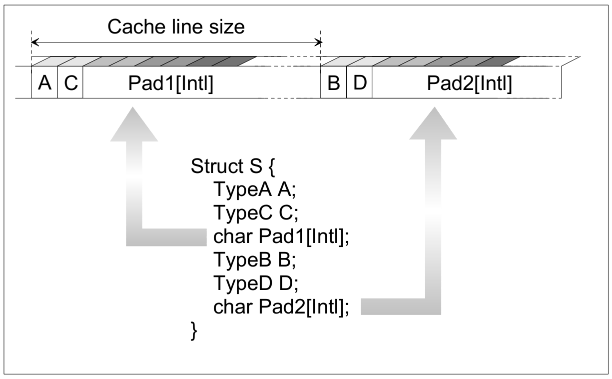

Figure 3. Interleaving is done by adding padding in the data structure declarations.

Implementation through structure declaration Instance interleaving was not foreseen when programming languages were defined. Therefore, they do not provide any support for it.

However, in C language data structure declarations may be tweaked to enable instance interleaving. Figure 3 illustrates this and how it affects the layout of an instance in memory (color code of the instances is the same as in Figure 2). A padding array is inserted after each interleaved chunk of the structure (chunks (A, C) and (B, D)).

Our goal is to reuse the unused space within the padding arrays to allocate and interleave the chunks of other instances of the structure. Every chunk must have the same size to allow the compiler to manage address computation. The set of least frequently used fields of the structure is divided into several equally sized chunks as needed.

Allocation within the padding structure is not directly supported by the C language. The new memory allocation library called *ialloc*, presented in the next section was developed for this purpose.

3. Ialloc: an allocation library for instance interleaving

Instance interleaving is natural neither for declaration nor allocation for programming languages like C. Moreover, instance interleaving in likely to be used as a final performance tuning step of an application development: only minimum source code modification is then tolerated.

In order to assist the developer in using instance interleaving in this final performance tuning step, we developed a specific allocation library, called *ialloc*. *Ialloc* hides from the user most of the complexity of instance interleaving management while its use incurs very limited and localized source code modification.

In this section, we present the implementation of the *ialloc* library and its current limitations.

3.1. Implementation

The data structures that need to be allocated with *ialloc* are only those which are most frequently used and which have also the most instances allocated.

Ialloc allocates arenas Our library *ialloc* reserves and manages special areas in the heap called arenas for each specific data structure it handles. An arena is a contiguous portion of the heap reserved for the allocation of specific data sizes. The concept of arenas has already been used in the Gnu-C allocation library to improve the locality of the library [10].

Different structures are likely to have different interleaving parameters. Therefore, an arena only holds instances of a single data structure. We interleave a sufficient number of frequently used chunks to fill the cache size, so the size of the arena is a multiple of the cache size. Each arena is managed independently with a small header holding its allocation table.

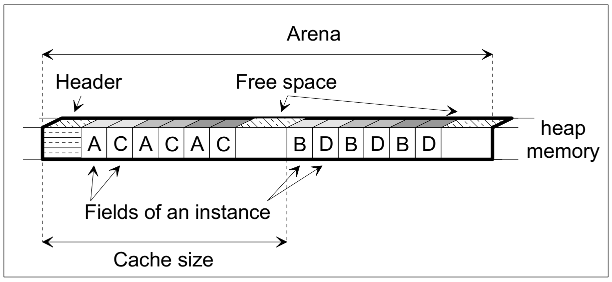

Figure 4. Inner organization of an allocation arena.

Figure 4 illustrates this. The two chunks (A, C) and

(B, D) of the structure are interleaved. The arena is almost[1] twice the size of the cache. The data structure is padded with arrays, each slightly smaller than the cache size.

An arena usually holds a few thousand instances. Therefore it is usually not large enough to accommodate all the instances of a structure. The *ialloc* library manages a chained list of arenas for each interleaved data structure. More arenas are allocated as needed. Each structure type is assigned an index corresponding to its own chained lists of arenas.

3.2. Using ialloc

To use the *ialloc* library, the programmer first modifies the data structure declaration to reorganize the fields into chunks and insert the padding arrays. Afterwards the user replaces corresponding calls to *malloc()* and *free()* by calls to *ialloc()* and *ifree()*, *ialloc*'s allocation routines. Figure 6 shows the transformation necessary from the original code sample shown in figure 5.

```
struct S{
        struct S * A;
        int B;
        int C;
        int D;
};

struct S *SPtr;
..
SPtr = (struct S *) malloc(sizeof(S));
SPtr→A = NULL;
if (..) SPtr→B= ..
free(SPtr);
..
```

Figure 5. Sample original code.

Except for the calls to *malloc* and *free*, all the references to the structure are left unchanged.

To pass static structure layout information to the allocation library, we need to provide three arguments to *ialloc()*. We give the type *S* of the structure[2] to tell it the size of the structure and to handle casting automatically, the size *ChunkSizeS* of the interleaved chunks, and the index *IndexS* chosen for that data structure type.

Adding padding in the declaration of the data structure forces the compiler to generate correct addressing to the various fields of the padded structure. The *ialloc* library hides the memory management and pointer arithmetic necessary to allocate non-contiguous chunks within the padding of the other instances.

[1]The header doesn't need to be duplicated.

[2]*ialloc()* is actually a *define*, which accepts directly the data type to prevent the need to cast

```
#define IndexS 0
#define ChunkSizeS sizeof(A)+sizeof(C)
struct S{
        struct S * A;
        int C;
        Pad1[IntlS];
        int B;
        int D;
        Pad2[IntlS];
};

struct S *SPtr;
..
SPtr = ialloc(S,ChunkSizeS,IndexS);
SPtr→A = NULL;
if (..) SPtr→B= ..
ifree(SPtr);
..
```

Figure 6. Sample optimized code.

These two optimization steps cannot be directly hidden in C, because allocation must be handled dynamically by a library while data structure layout and pointer arithmetic is generated statically by the compiler.

Usage constraints The code transformation process shown above is valid for recursive data structures (i.e. structures organized in chained lists, trees...). However, some complications may arise, because static and dynamic layout optimizations must interact and C Language doesn't provide any support for that. Therefore layout optimization cannot be transparent to C programmer.

We list some limitations when using *ialloc* below:

▷ *ialloc* does not currently support arrays of data structures. The difficulty lies in how pointer arithmetic is done by the C compiler. Let's consider a reference to an element of an array of structure like $Tab[i].A$. Using our approach to interleave this structure would necessitate extensive casting to allow correct array pointer arithmetic. This would mean heavy code transformations.

▷ Allocation of different types of structures at the same *malloc()* call site cannot be handled directly, because our library asks for an index for each structure type allocated. This problem happens when *malloc()* is called within a wrapper function, for example when the programmer handles structure allocation himself or when he checks the correctness of the allocation process. This is the case in the sources of the Gnu-C compiler. The wrapper function must be removed before optimization.

▷ Freeing of different structure types by the same *free()*. The *free()* can be converted to *ifree()* only if all instances freed are allocated by the *ialloc* library.

▷ Interleaved structures cannot support assignations between instances without compiler support. Fields must be copied explicitly one by one, i.e , if $P1$ and $P2$ are pointers to interleaved instances of a structure, "$*P1 = *P2$;" should be converted to "$P1 \rightarrow A = P2 \rightarrow A; P1 \rightarrow B = P2 \rightarrow B$;..'. This type of assignation is unlikely to be encountered in C, since it is legal only in ANSI-C compilers. However it is encountered in C++ programs to initialize objects, but we can always overload the equal operator of the interleaved classes.

▷ Static allocation of interleaved structures should be avoided since padding arrays use up a lot of space. If the allocation library is bypassed, the padding space is wasted. A solution is to convert the static instances of the structure into dynamically allocated instances. However we never met this difficulty in the experimentation we did, since static allocation is rarely mixed with dynamic allocation.

In our experiments, we found that the biggest difficulty with our *ialloc* library is its inadequation to handle directly the allocation of arrays of structures. Their use is quite frequent in large applications. The usage of wrapper allocation functions is also occasionally a nuisance.

4. Experimentation

4.1. The target architecture

The programs benchmarked in this section were run as monoprocessor tasks on an SGI Origin 2000 with 195MHz R10000 processors. The L1 on-chip data cache is a 2 way set-associative non-blocking cache of 32KB, with 32B lines. The L2 external cache has the same configuration but is 4MB. The local memory of the node is made of 128MB of SDRAM.

Statistics reported were gathered using SGI's *Speedshop* package. It can count misses in all levels of the memory hierarchy using hardware counters and computes an estimation of the corresponding time penalties incurred.

4.2. Benchmark selection

Data layout optimizations are intended to provide performance gains on applications with poor data locality and large working sets. SpecInt95 benchmarks for instance do not correspond to this class of applications. Therefore, we gathered a few applications from different sources that make extensive use of dynamically allocated recursive data structures (lists, Btrees, quad-trees..). The selected applications spend between 15% and 80% of their user time waiting for memory accesses.

We chose Health and Tsp, two Olden benchmarks [18, 5] also used to study data structure prefetching [14], Jigsaw, a WPI benchmark used to test virtual memory swapping of the MACH OS kernel [7, 8], Raytrace, a program from the Splash-2 parallel benchmark suite [22], and Radiosity, a program developed at INRIA by Meneveaux et al. [15].

Characteristics of these applications are further detailed below. For each application, we also present the optimizations that were performed on the data layout.

▷ Health : "health 6 500"
Program simulating the Colombian health care management. A simulation of patients transfers between the hospitals of 1365 villages ($\sum_{n=0}^{n=6-1} 4^n$) during 500 time cycles. Four structures are used, *village* (a quad-tree), *hospital*, *patient* (patient's data), and *patient list* (nodes of a chained list). We interleave the last two data structures fields by fields: those have the most instances and are most often referenced. The program consumes 17MB of memory.

▷ Tsp : "tsp 3276800"
Traveling salesman problem solved using the closest point heuristic. The problem size is the traversal of over 3.27 million cities. We interleave the only data structure used (a quad-tree) by chunks of eight bytes. Doubles are alone, but pointers are grouped two by two. The program uses 193MB of memory.

▷ Jigsaw : "echo 100 | jigsaw 100.out"
Solves a mathematical representation of a 100x100 2D-jigsaw puzzle. Tiles are represented by a structure with 11 word-sized fields. Tiles are stored in a chained list. We interleave the tiles field by field. The program uses 3.5MB of memory.

▷ Raytrace : "raytrace balls4.env"
Virtual image rendering using raytracing. We use the balls4.env scene made of 7381 spheres reflecting on one another and organized into a fractal shape. We modified the balls4.env file to add "hu_gridsize 1", to make the application run longer. Many structures are used in this program. We interleaved the structure *object* field by field and the structure *sphere* by regrouping 5 fields and separating the last one. The program uses 40MB of memory.

▷ Radiosity : "radiosity -i tt.gra -o tt -T 10 -A 1 -E 0.001 -N 1500 -s -m"
Virtual image rendering using hierarchical radiosity. The program is built to render extremely large scenes. The input is a scene made of 213 lightly furnished rooms, each one is described in a separate *nff* file. This program uses many structures. We interleave three structures field by field: *List_surfaces*, *List_patches* and *List_links*. The program uses 127MB of memory.

The programs are compiled using the C MipsPro compiler, with the most aggressive optimizations: "cc -Ofast=IP27 -LNO -IPA", except when profiling with *ssrun* which does not support aggressive optimizations. Raytrace was compiled with "cc -O3 -n32 -R10000 -mips4" because results are wrong when optimizations are too aggressive.

4.3. Optimization process

As a first step, profiling is used to determine source code lines generating large number of cache misses. We use *Speedshop*'s *ssrun* and *prof* tools. Optimization of the data structure(s) referenced by these code lines is done manually. That is manual search of the most referenced fields using the profiles, reorganization and interleaving of the structures and replacement of *mallocs* and *free*.

4.4. Performance Results

Applying our data layout enhancements leads to significant performance increase as illustrated on Figure 7.

The benchmarked applications have either poor L1, L2 or TLB performance with their original layout as indicated in the figure. For each benchmark, the miss reduction is illustrated for the components of the memory hierarchy that have a significant influence on the execution time. These miss reductions can be quite dramatic: for instance miss ratios are improved by over 90% for Radiosity and Jigsaw. These two programs declare large data structure and use a small number of fields in each instance. In that case cache line space is completely wasted.

Speedups ranging from 1.08 to 2.52 were observed. Notice that high speedups are obtained when the bottleneck is the TLB or the L2 cache, where a miss costs from 68 to 75 cycles[3].

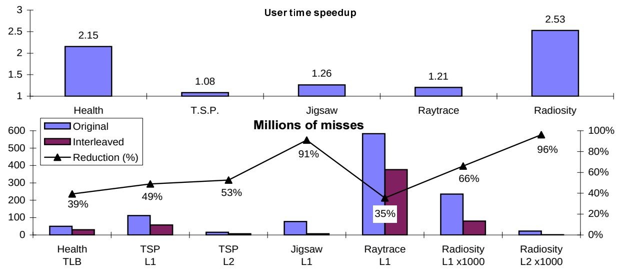

Figure 7. Data layout provides significant cache miss reduction which induce speedups.

Other performance factors Changing the data layout by the usage of the *ialloc* library slightly changes the type and the number of instructions executed. Figure 8 shows that this variation is minor, therefore it cannot affect the execution time significantly for our benchmarks.

By interleaving data in memory, we also tend to save up memory compared to malloc. Using arenas is more cost-effective than tying an 8-byte tag to each allocated block, as is usually done by standard allocation libraries. For example, 34.73% of memory is saved on *health*, figure 8.

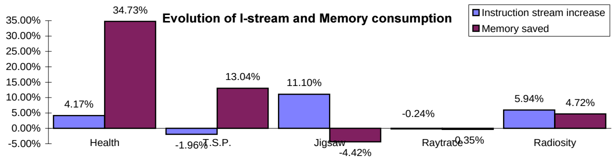

Figure 8. ***ialloc*** **slightly increases the I-stream and generally save memory over malloc.**

4.5. Independence of the input set

The speedups and miss ratio improvements reported previously were measured using the same input set for both profiling and optimizing. This may lead to biased results if other input sets do not have the same dominant access patterns on data structures.

In this section, we analyze the sensitivity of our data layout optimizations to the input sets.

Our intuition is that the optimizations we propose should be valid for any input set, because the global reference patterns of an algorithm over a data structure in an application is likely to be very similar with different input sets. Even if the data stored in the structures change, the portions of code used to access the instances of the structure do not change. Since field references are statically encoded in the program by the compiler, the optimized structure layout is also likely to be adapted to other data sets for the profiled routines.

Input sets To verify this assertion, we tried several different input sets for all our applications. Input parameters were changed for Health and TSP, while for Jigsaw, Raytrace and Radiosity we changed the input files.

For Jigsaw, we used different puzzle files, generated automatically by a separate program provided with the benchmarks. The 64x64 puzzle is the default puzzle of the benchmark.

For Raytrace we built a new scene with over 10000 spheres organized as two planes facing each other. The rays shooting should have a completely different behavior than with balls4.env because the scene is organized quite differently.

For Radiosity we used a scene with 4 classrooms facing a patio. This scene holds 5 rooms instead of 213, but each one is much more furnished (more chairs, tables, and lights...).

Experimental results We first check for the programs that the memory bound routines do not radically change with other input sets. For example with *Radiosity*, three

[3]These cost estimations are those used by *SGI*'s *perfex* tool.

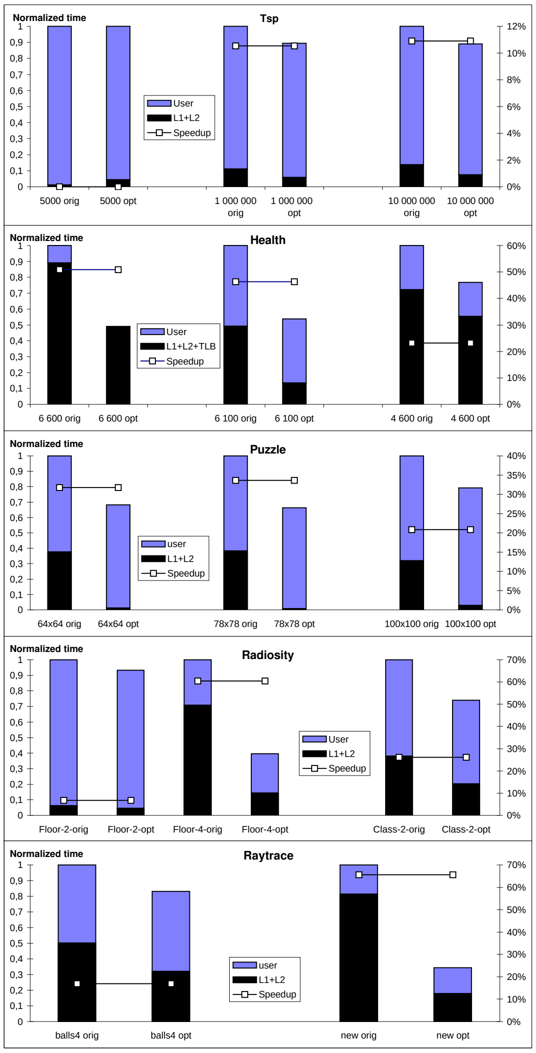

Figure 9. Speedup with different input files.

source lines generate 79.7% and 92.8% of the misses respectively for the *Balls4* and *New* input files.

Figure 9 illustrates speedups for all the benchmarks. Estimated time lost on cache misses is also illustrated. Time is normalized, because runtimes vary from less than a second to days, depending on the application and the input set.

Results are consistent. The optimizations increase the performance by reducing the time lost in the memory hierarchy. Furthermore, the larger the working set and the execution time, the higher the benefit from data layout optimizations.

5. Related works

Scalar data layout optimizations were studied in [21, 16]. Many studies surveyed in [1] have addressed numeric array layout optimizations (padding) and computation reordering (loop transformations) to improve cache behavior.

On the other hand, few studies have addressed improving the cache behavior of applications using heterogeneous data structures.

Prefetching has been studied for self-referencing data structures [14, 13]. Prefetching can improve global performance of the applications. However it can also saturate memory bandwidth and finally become counter-productive. Our approach is *a contrario* to improve the spatial locality of the program.

Most works to improve the performance of applications using dynamically allocated data have focussed on optimizing the allocation libraries to provide efficient memory management. The first approach was to develop libraries managing memory reuse with efficient algorithms [20]. Zorn et al. analyzed the locality of dynamic allocation libraries [10]. These codes often exhibit poor locality. New allocations libraries such as *Customalloc* [9] were proposed. These studies focus on locality inside the allocation library, but do not consider the subsequent accesses of the program to the allocated data structures.

6. Conclusion

Delays wasted accessing the memory hierarchy are impairing the performance on many applications. This difficulty is likely to increase in the next decade [19, 3]. To maximize cache performance, software locality optimization techniques have been shown to be very efficient on applications working on numerical arrays [2]. However, many applications do not use data arrays, but heterogeneous data structures.

Two techniques may be used for improving data layout. Field reorganization consists in regrouping together the most frequently used fields of a structure to fit them in a single cache line. This may reduce cache line space wasted. However there are generally only a few fields frequently used in a structure, not enough to fill a cache line. Therefore we propose instance interleaving. It consists in grouping the most frequently used fields of several instances to fit them into the same cache line.

The first contribution of this paper is to show that combining field reorganization with instance interleaving can be an efficient way to improve the memory behavior and the overall performance of the application. On our benchmark set of five C applications making a heavy usage of dynamically allocated data structures, speedups ranging from 1.08 to 2.53 are obtained. Miss ratios (L1, L2 or TLB depending on the application) are reduced by 35% to 96%.

However, instance interleaving has no natural language support in C. The second contribution of this paper is the design of *ialloc*. It is a dedicated memory allocation library built to support instance interleaving of dynamically allocated structures. When using *ialloc*, optimizations can be implemented on C code with minimum source changes: only structure declarations, *free()* and *malloc()* functions are touched.

At present, the automatic detection of the most frequently used fields of a structure is beyond the possibility of current compiler technology. Therefore, data layout optimization must be done by the programmer by profiling the applications to detect and regroup the most frequently used fields of the data structures. This might be seen as a major limitation. However few structures need to be interleaved in a program, since few structure influence the program's reference patterns [10]. Furthermore, from our experiments we find that the most frequently used fields of the data structures do not change when we run the application with different input sets. We think that an *ialloc*-like allocation library associated with some specific profiler dedicated for heterogeneous data structures might be of great help in a performance tuning toolbox for general applications.

Acknowledgements We would like to thank M. Carlisle for giving us access to his benchmarks, as well as D. Meneveaux for his radiosity program.

References

[1] D. F. Bacon, J.-H. Chow, and D. R. Ju. A compiler framework for restructuring data declarations to enhance cache and tlb effectiveness. *Proceedings of CASCON'94*, Nov. 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] D. Burger, J. R. Goodman, and A. Kägi. Quantifying memory bandwidth limitations of current and future microprocessors. *23rd International Symposium on Computer Architecture*, 1996.

[4] D. C. Burger, J. R. Goodman, and A. Kägi. The declining effectiveness of dynamic caching for general purpose multiprocessor. Technical Report 1261, University of Winsconsin, http://www.cs.wisc.edu/g̃alileo, 1995.

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

[6] J. J. Dongarra and D. W. Walker. Software libraries for linear algebra computation on high performance computers. Technical Report ORNL TM-12404, Oak Ridge National Laboratory, http://www.nhse.org/hpc-netlib/, Aug. 1993.

[7] D. Finkel, R. E. Kinicki, A. John, B. Nichols, and S. Rao. Developing benchmarks to measure the performance of the mach operating system. In *USENIX Mach Workshop*, pages 83–100, 1990.

[8] D. Finkel, R. E. Kinicki, J. A. Lehmann, and J. CaraDonna. Comparison of distributed operating system performance using the wpi benchmark suite. Technical Report WPI-CS-TR-92-2, Worcester Polytechnic Institute, 1992.

[9] D. Grunwald and B. Zorn. Customalloc: Efficient synthetized memory allocators. *Software Practice & Experience*, 23(8):851–869, Aug. 1993.

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

[11] A. S. Huang and J. P. Shen. A limit study of local memory requirements using value reuse profiles. *28th Annual International Symposium on Microarchitecture*, 1995.

[12] A. S. Huang and J. P. Shen. The intrinsic bandwidth requirements of ordinary programs. In *7th symposium on Architectural Support for Programming Languages and Operating Systems*, page 105, 1996.

[13] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. Spaid: Software prefetching in pointer and call-intensive environments. *28th Annual International Symposium on Microarchitecture*, 1995.

[14] 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.

[15] D. Meneveaux, K. Bouatouch, and E. Maisel. Memory management schemes for radiosity computation in complex environment. PI 1097, IRISA, ftp.irisa.fr/techreports/1996/PI-1097.ps.gz, 1997.

[16] P. R. Panda, N. D. Dutt, and A. Nicolau. Memory data organization for improved cache performance in embedded processor applications. *ACM Transactions on Design Automation of Electronic Systems*, 2(4):384–409, Oct. 1997.

[17] D. Patterson and J. Hennessy. *Computer Architecture: A Quantitative Approach, 2nd ed.*, chapter 5, Memory hierarchy design. Morgam-Kaufman, 1996.

[18] 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.

[19] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the memory wall: The case for processor/memory integration. In *23rd International Symposium on Computer Architecture*, 1996.

[20] A. S. Tanenbaum. *Modern Operating Systems*, chapter 3. Morgan Kauffmann, 1995.

[21] D. N. Truong, F. Bodin, and A. Seznec. Accurate data layout into blocks may boost cache performance. Technical Report 1000, IRISA/INRIA, ftp.irisa.fr /techreports/1996/PI-1000.ps.gz, 1996. presented at the *Second Workshop on Interraction between Compilers and Computer Architecture*, San Antonio, TX, Feb. 1997.

[22] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In ACM, editor, *22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.