Automatic Restructuring of Linked Data Structures

H.L.A. van der Spek, C.W.M. Holm, H.A.G. Wijshoff

Leiden University, LIACS,
Niels Bohrweg 1, 2333 CA Leiden,
The Netherlands

Abstract. The memory subsystem is one of the major performance bottlenecks in modern computer systems. While much effort is spent on the optimization of codes which access data regularly, not all codes will do so. Programs using pointer linked data structures are notorious for producing such so called irregular memory access patterns. In this paper, we present a compilation and run-time framework that enables fully automatic restructuring of pointer-linked data structures for type-unsafe languages, such as C. The restructuring framework is based on run-time restructuring using run-time trace information. The compiler transformation chain first identifies disjoint data structures that are stored in type-homogeneous memory pools. Access to these pools is traced and from these run-time traces, a permutation vector is derived. The memory pool is restructured at run-time using this permutation, after which all pointers (both stack and heap) that refer to the restructured pool must be updated. While the run-time tracing incurs a considerable overhead, we show that restructuring pointer-linked data structures can yield substantial speedups and that in general, the incurred overhead is compensated for by the performance improvements.

Key words: Restructuring compilers, linked data structures

1 Introduction

Predictability in memory reference sequences is a key requirement to obtain high performance on applications using pointer-linked data structures. This contradicts the dynamic nature of such data structures, as pointer-linked data structures are often used to represent data that dynamically changes over time. Also, different traversal orders of data structures cause radical differences in behavior.

Thus, having control on data layout is essential for getting high performance. For example, architectures like the IBM Cell and GPU architectures each have their own characteristics and if algorithms using pointer-structures are to be executed on such architectures, the programmer must mold the data structure in a suitable form. For each new architecture, this means rewriting code over and over again. Another common pattern in code using pointer-linked data structures is the use of custom memory allocators. Drawbacks of this approach are that

such allocators must be implemented for various problem domains and they depend on the knowledge of the programmer, not on the actual behavior of the program. Our restructuring framework is a first step in the direction to liberate the programmer from having to deal with domain specific memory allocation and rewriting of data structures.

In this paper, we present a compiler transformation chain that determines a type-safe subset of the application and enables *run-time* restructuring of type-safe pointer-linked data structures. This transformation chain consists of type-safety analysis after which disjoint data structures can be allocated from separate memory pools. At run-time, accesses to the memory pools are traced temporarily, in order to gather actual memory access patterns. Next, from these access patterns, a permutation is generated which enables the memory pool to be reordered. Note that these traces are not fed back into a compiler, but are rather used to restructure data layout at *run-time* without any modification of the original application. Pointers in the heap and on the stack are rewritten if the target they are pointing to has been relocated. After restructuring, the program continues using a new data layout.

Restructuring of linked data structures cannot be performed unless a type-safe subset of an application is determined. This information is provided by Lattner and Adve's Data Structure Analysis (DSA), a conservative whole-program analysis reporting on the usage of data structures in applications [6, 8]. The analysis results of DSA can be used to segment disjoint data structures into different memory regions, the memory pools. Often, many memory pools turn out to be type-homogeneous, i.e. they store only data of a specific (structured) type. These kind of pools are our starting point.

For type-homogeneous pools, we have implemented *structure splitting*, similar to MPADS, the memory-pooling-assisted data splitting framework by Curial et al. This changes the physical layout of the structures, but logically they are still addressed in the same way. Structure splitting is not a strict requirement for restructuring, but it simplifies the implementation and results in higher performance after restructuring.

Tracing does have a significant impact on performance, so we allow that tracing can be disabled after optimizing access to a memory pool. The application itself does not need to be aware of this process at all. While in principle such a trace can also be used to modify the behavior of a memory allocator for the next execution of an application, we have not done so at this moment. It is important to note that tracing and restructuring all happen within a single run of an application.

In order to illustrate the need for restructuring, it is interesting to have a look at what could potentially be achieved by controlling data layout. For this, we used SPARK00, a benchmark set in which the initial data layout can be explicitly controlled. Figure 1 shows the potential speedups on an Intel Core 2 system (which is also used in the other experiments, together with its successor, the Core i7) if the data layout is such that the pointer traversals result in a sequential traversal of the main memory, compared to a layout that results

in random memory references. This figure illustrates the potential for performance improvements if data layout could be optimized. Our framework intends to exploit this potential for performance improvements.

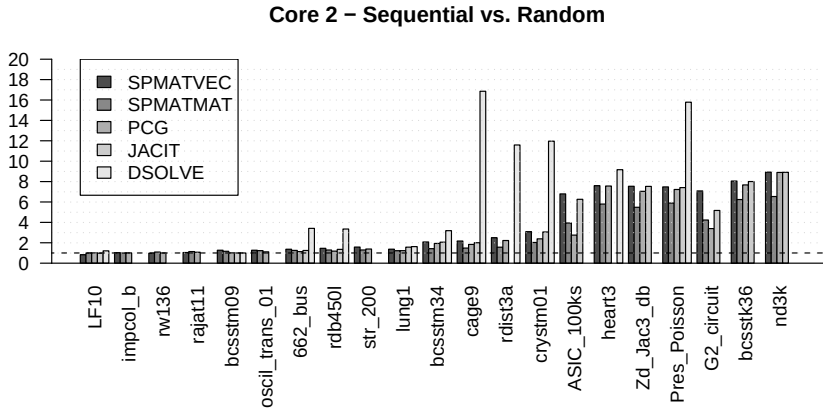


Fig. 1. Speedup when using data layout with sequential memory access vs. layout with random memory access.

In Section 2, our framework is discussed in detail. Section 3 contains the experimental evaluation of our framework. Restructuring pointer-linked data structures has great potential and in this paper considerable speedups are shown on the SPARK00 benchmarks. The challenge of SPARK00 lies in closing the performance gap between random access behavior and perfectly sequential access behavior. As such, it illustrates the potential, but it does not guarantee that such speedups will be obtained for any application. The overhead of tracing mechanism, which of course does not come for free, is discussed in Section 3.2. It is shown that the performance gains do compensate for this overhead within relatively few consecutive uses of the restructured data structure. Related work is discussed in Section 4. Future work and conclusions are given in Section 5.

2 Restructuring Pointer-linked Data Structures

In this section, the restructuring framework for pointer-linked data structures is described. First, the type-safety analysis pass we use is described along with pool allocation and structure splitting of data in such pools. We also describe how pointers to memory pools that live on the stack are tracked, so they can be rewritten at run-time. Next, the analysis of accesses to such pools is described. Using this analysis, tracing code is generated which traces accesses to pools at the field granularity, at run-time. Eventually, we describe how from these traces

a permutation vector is computed, which is used to reorder a pool and update all references to such pools.

2.1 Data Structure Analysis and Pool Allocation

Lattner and Adve's Data Structure Analysis (DSA) is an efficient, interprocedural, context- and field-sensitive pointer analysis [5, 6, 8]. It is able to identify (conservatively) disjoint instances of data structures even if these data structures show an overlap in the functions that operate on them. The analysis can proceed even if information on the application is incomplete, which for example is the case if external libraries are used. DSA has been implemented in the LLVM compiler framework [7] which is especially designed to handle optimization throughout the entire lifetime of the application. Our optimizations are performed after linking the application, such that a *full program view* is available with the exception of calls to the standard C library.

```
int main( int argc, char **argv )
{
    ...
    MatrixPtr tmp = ReadMatrixPtrRow( matrixFile );
    MatrixPtr Matrix = MatrixToFormat( tmp, format );
    ...
    for( i = 0; i < iterations; i++ )
        MatrixMultiplyVec( Matrix, right, result );
    ...
}
```

Fig. 2. Code excerpt of main function of SPMATVEC.

DSA generates a data structure graph (DSGraph) for each function. This DSGraph describes the data structures taking into account the effects of the associated function and all its callees. Figure 2 shows a part of the main function of SPMATVEC, one of the benchmarks used in the evaluation of our method (see Section 3). Figure 3 shows the associated DSGraph. Information about the variables generated by the compilation to the LLVM bitcode (which uses an SSA representation) are not shown. The graph shows the two stack variables (specified by the S flag) $\%tmp$ and $\%Matrix$, which both have their disjoint storage space on the stack. Hence the separate nodes. The $MatrixFrame$ structure they are both pointing to is one node, indicating that the analysis cannot prove that they are pointing to disjoint structures. The $MatrixFrame$ structure basically contains three pointers. These are the three arrays of pointers that point to the start of a row, the start of a column and the diagonal elements. The $MatrixElement$ structure is the structure containing the matrix data. It has two self references, which are the two pointers used to traverse the matrix row- and column-wise.

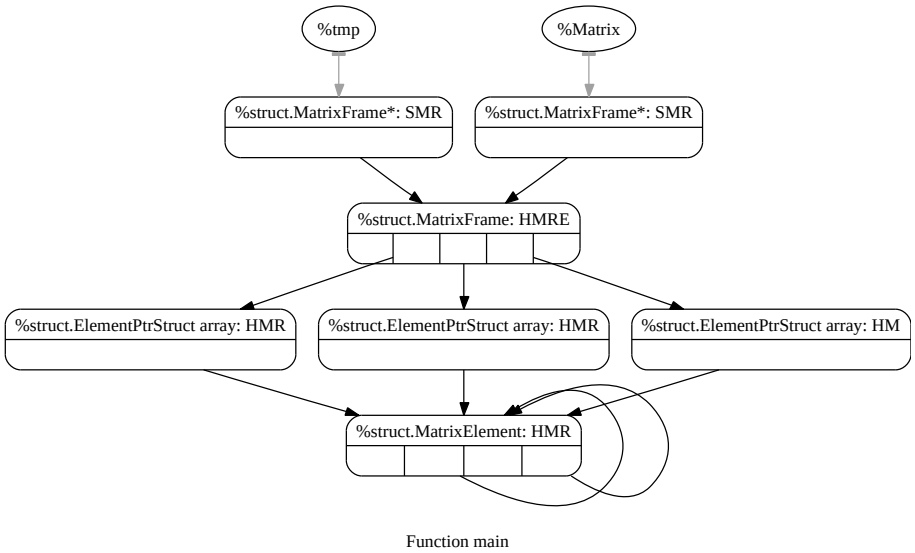


Fig. 3. DSGraph for main function of SPMATVEC benchmark.

Each of the nodes in the graph can utilize its own memory pool. If a node is type-homogeneous, its structures can be split and the contents of such a pool can be permuted at run-time. In this paper, restructuring will be done on the pool containing the $MatrixElement$ structures.

The layout of structure elements in type-homogeneous pools can be remapped, as access to such pools can be described in terms of an *object identifier* and a *field number*. An obvious way to remap structure layout is to store structures grouped by field, which is called structure splitting. Conceptually, this converts an array of structures into a structure of arrays. Our implementation of structure splitting is similar to the MPADS framework of Curial et al. [2]. In addition, our implementation supports nested structures, by moving the fields from the nested structure into the outer structure. Nested arrays of structures are not supported. Any data access to a split pool can thus be viewed as access to an object identifier/field pair, even for nested structures.

Structure splitting effectively is nothing more than identifying access to a pool, and redefine the semantics of the computation of pointers into that specific pool. In LLVM, all address calculations are performed using the *GetElementPtr* instruction (further referred to as GEP instruction). As everything is transformed into unnested structures, all field accesses to a data structure element take the following form:

```
%reg = getelementptr %T* %p, i32 0, i32 fieldNr
```

Written in C, this is roughly equivalent to: $\&(p+0).field$. Note that in LLVM, $fieldNr$ must be a compile-time constant integer defining the field number. All

GetElementPtr instructions that compute pointers into pools are registered and in a later pass, these instructions are transformed into explicit pointer computations according to the desired split layout.

2.2 Pool Access Analysis

The LLVM representation itself does not reason in terms of pools and fields. Therefore a pool access analysis pass is needed which determines for each memory operation the following properties: the pool descriptor (a unique, per pool pointer to a structure defining the properties of the pool), the object identifier and the field. Consider the following LLVM code snippet:

```
; Load pElement->Real;
%tmp = load %struct.MatrixElement** %pElement, align 8
%tmp2 = getelementptr %struct.MatrixElement* %tmp, i32 0, i32 2
%tmp3 = load double* %tmp2, align 8            ; Pool access
```

*tmp*2 is a pointer to a field of some object in a pool. When considering a load, the analysis first looks for the underlying object, which is defined by the first operand of the defining GEP instruction. For this underlying object, a mapping to its corresponding pool descriptor has been determined by the structure splitting analysis pass. The field is given by the last operand of the GEP instruction.

2.3 Pointer Tracking

The fact that we actually permute the actual data layout of memory pools at run-time implies that all references to such a pool must be kept track of. For pointers stored in the heap connectivity information is provided by DSA. This connectivity information is made available to the run-time environment, and thus all references to a pool on the heap can be identified. For pointers stored on the stack the location of these pointers in memory must be known in order to update these pointers when their target has been relocated. This is done by generating code at compile-time that registers all pointers to pools on the stack to the run-time. After restructuring, all references on the stack are known and can be updated.

2.4 GetElementPtr Instruction Rewriting

The regular LLVM code generation backend is not aware of any alternative data layout mappings. Therefore, any address calculations must be transformed into explicit pointer arithmetic. All GEP instructions that generate pointers to pools are now remapped such that the split layout is used.

Pointers that reside on the heap and that point to objects (not to their fields) in split pools are stored as object identifiers to make the representation position independent. Object pointers that reside on the stack are stored as

regular pointers, and so are derived pointers (pointers to fields) on the stack. Due to the properties of DSA, no derived pointers exist on the heap.

Pointers that need to be stored as object identifiers are converted to object identifiers before they are stored to the heap as follows: $objid = \frac{(objaddr - pool\_base)}{sizeof(firstfield)}$ Whenever such a pointer is load from memory, it is directly converted back to a regular pointer: $objaddr = pool\_base + objid \times sizeof(firstfield)$

2.5 Memory Access Tracing

In order to restructure a memory pool a permutation must be supplied to the restructuring run-time. The pool access analysis pass provides the information about all memory references and these memory references can all be traced. Traces are generated per pool, per field. For each pool/field combination, this results in a trace of object identifiers. From any of these traces, a permutation vector can be derived which can be used to permute a pool. The permutation vector is currently computed by scanning the trace sequentially and appending the object identifiers encountered to the vector, avoiding duplicates.

Tracing does not come for free and therefore tracing should be avoided if it is not necessary. For the evaluation of our restructuring method we choose to trace the first execution of a specified function (compiler option), restructure using this trace and then disable tracing. In a future implementation, this will be dynamic and tracing could be triggered if a decrease in performance is detected (for example by using hardware counters).

2.6 Run-time Pool Restructuring

The memory tracing mechanism for which code is generated produces per pool, per field traces at run-time, if tracing is enabled. Using these traces, a permutation vector can be generated according which is used to remap all allocated elements in a pool. This remapping consists of rewriting the pool that must be restructured, updating all referring pools such that all fields containing pointers to the restructured pool are updated, and updating all pointers (both to the objects as well as fields of objects) to the restructured pool. There are no pointers from other pools to fields of a restructured object. This is a result of the type-safety properties required by Data Structure Analysis.

During restructuring, each element of each field is copied to a newly allocated memory space to the position indicated by the permutation vector. Next, all pointer fields of all referring pools (including self references) are updated using the permutation vector. In Figure 3, if the pool for *MatrixElement* would be restructured, two self-referring fields have to be rewritten as well as three fields from other pools. Finally, all pointers on the stack that have been registered by the pointer tracking mechanism are updated as well.

3 Experiments

The challenge of a restructuring compiler is to generate code that will automatically restructure data, either at compile- or run-time in order to achieve performance that matches the performance when an optimal layout would be used. In the introduction the potential of restructuring was shown by comparing execution of the benchmarks using explicitly defined data layouts. In the experiments here, we ideally want to obtain similar performance gains, but then by automatic restructuring of data layout of the used pointer-linked data structures.

We use the benchmark set SPARK00 which contains pointer benchmarks whose layout can be controlled precisely [13]. The pointer-based benchmarks used are: SPMATVEC (sparse matrix times vector), SPMATMAT (sparse matrix times matrix), DSOLVE (direct solver using forward and backward substitution), PCG (preconditioned conjugate gradient) and JACIT (Jacobi iteration).

These benchmarks store their matrix using orthogonal linked lists (elements are linked row-wise and column-wise). All of them traverse the matrix row-wise, except DSOLVE, which traverses the lower triangle row-wise and the upper triangle column-wise.

For all benchmarks, one iteration of the kernel is traced, after which the data layout is restructured. After this, tracing is disabled. This all happens at run-time, without any hand-modification the application itself.

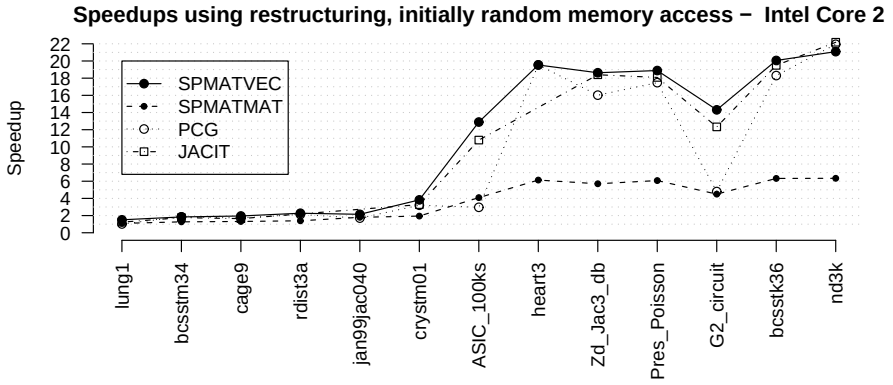
The experiments have been run on two platforms. The first is the Intel Core 2 platform, an Intel Xeon E5420 2.5 GHz processor with 32 GiB of main memory, running Debian 4.0. The other system is an Intel Core i7 920 based system with 6 GiB of main memory.

3.1 The Effect of Restructuring

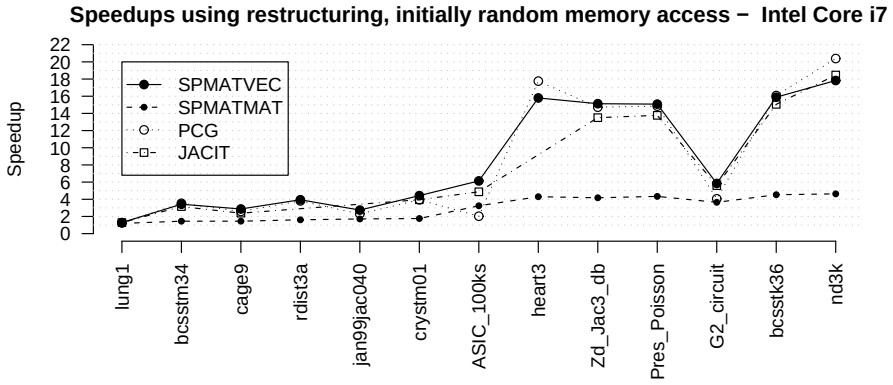
As shown in the introduction, being able to switch to an alternative data layout can be very beneficial. We applied our restructuring transformations to the SPARK00 benchmarks and show that in ideal cases, speedups exceeding 20 are possible by regularizing memory reference streams in combination with structure splitting. Of course, the run-time introduces a considerable amount of overhead and is a constant component in our benchmarks. We will consider this overhead separately in Section 3.2 to allow a better comparison between the different data sets.

Figure 4(a) and 4(b) show the results of restructuring on the pointer-based SPARK00 benchmarks (except DSOLVE, which is treated separately), if the initial data layout causes random memory access, on the Intel Core 2 and Core i7, respectively. The data set size increases from left to right. As shown in previous work [13], optimizing data layout of smaller data sets is not expected to improve performance that much and this fact is reflected in the results. On both architectures, restructuring had no significant effect for data sets fitting into L1 cache. These sets have not been included in the figures. For sets fitting in the L2 and L3 cache levels, speedups of $1 - 6\times$ are observed. The Core i7 has a 8 MiB L3 cache, whereas the Core 2 only has two cache levels. This explains the difference

in behavior for the matrix *Sandia/ASIC_100ks*, which shows higher speedups for the Core 2 for most benchmarks. However, it turns out that the Core i7 runs almost 3× faster when no optimizations are applied on SPMATVEC for this data set. Therefore, restructuring is certainly effective on this dataset, but the greatest benefit is obtained when using data sets that do not fit in the caches.



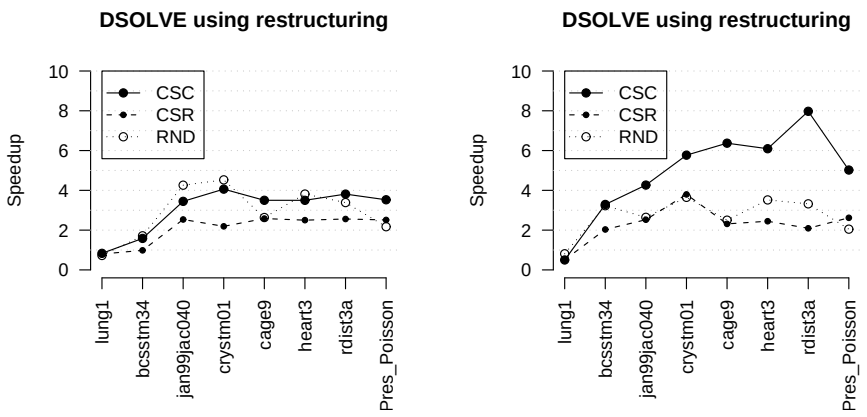
(a) Intel Core 2



(b) Intel Core i7

Fig. 4. Speedups obtained using restructuring on the SPARK00 benchmarks. The initial data layout is random.

An interesting case is DSOLVE, in which the lower triangle of the matrix is traversed row-wise, but the upper triangle is traversed column-wise. As the available data layouts of the matrices are row-wise sequential (CSR), column-wise sequential (CSC) or random (RND), none of these orders matches the traversal order used by DSOLVE. Figure 5(a) and 5(b) shows the results for DSOLVE using the different memory layouts on the Core 2 and Core i7, respectively.



(a) Intel Core 2 (b) Intel Core i7

Fig. 5. Speedups obtained using restructuring on DSOLVE for all different initial layouts. Input data sets are ordered by size (after LU-factorization).

The matrices are ordered differently than in the other figures, as DSOLVE uses LU-factorized matrices as its input, which have different sizes depending on the number of fill-ins generated during factorization. The matrices have been ordered from small to large.

For the *lung1* dataset, a decrease in performance is observed, but for the larger datasets, restructuring becomes beneficial again. Speedups of over 6× are observed for the Core i7, using CSC (column-wise traversal would yield a sequential memory access pattern) as initial data layout. In principle, the RND (initial traversal yields a random memory reference sequence) data set could achieve much higher speedups if after restructuring the best layout is chosen. Currently, this is not the case for DSOLVE and we attribute this to the very simple permutation vector generation algorithm that we use (see Section 2.5). Generation of permutation vectors from traces will be improved in future versions of the framework.

3.2 Tracing- and Restructuring Overhead

Our framework uses tracing to generate a permutation vector that is used to rewrite the memory pool. Traces are kept for each field of a pool and one of these traces is used for restructuring. Currently, the trace to be used is specified as a compiler option, but this could potentially be extended to a system that autonomously selects an appropriate trace. This will be addressed in a forthcoming paper.

Tracing and the subsequent restructuring step have an impact on the performance. One cannot simply trace everything all the time as the system will run out of memory very quickly. In the benchmarks, we choose to only trace the first

iteration of the execution of the kernel. In order to minimize the overhead of the tracing, the trace will only contain object identifiers, as described in Section 2.5. So for instance, if a linked list contains a floating point field and this list is summed using a list traversal, then both the pointer field and the floating point field are traced, there is an overhead of 2 trace entries per node visited. In our experiments, the structure operated on is 32 bytes and tracing above mentioned traversal would add 16 bytes per node extra storage requirements when using 64-bit object identifiers. Using 32-bit objects identifiers, this would be reduced to 8 bytes. Subsequently, the memory pool is restructured using the information of the trace which relates to the field that contains the floating point values of the linked list nodes.

The overhead of the tracing and restructuring has been estimated by running a single iteration of each kernel with and without tracing and restructuring enabled, using a data layout causing random memory access. Figure 6 shows

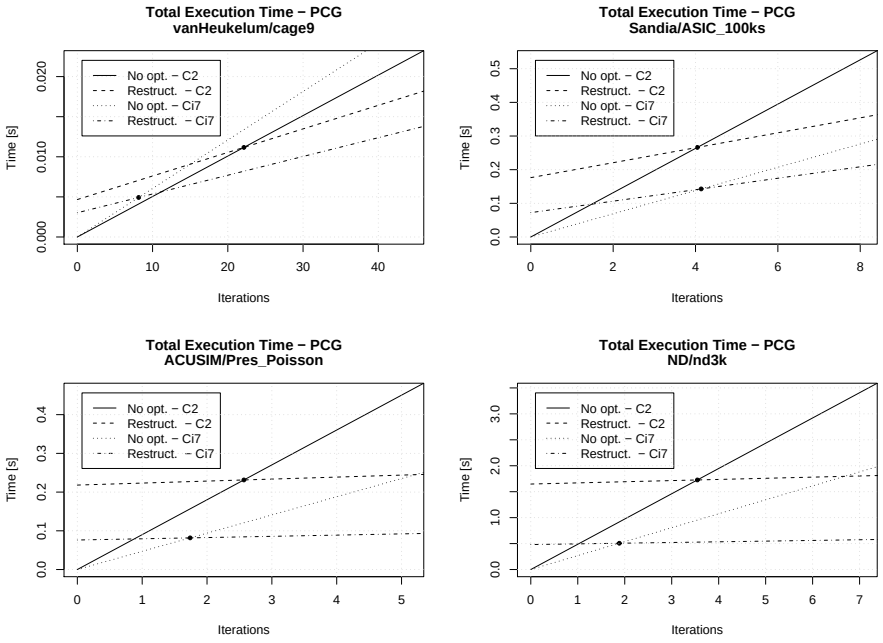


Fig. 6. Execution times with and without restructuring. The break-even points are marked with a dot.

the interpolated execution times of the benchmark PCG, both with and without restructuring for the Core 2 and Core i7 architectures. The initial data layout produces random memory access behavior of the application, which is eliminated after the first iteration when tracing and restructuring is used. After the first it-

	spmatvec		*spmatmat*		*pcg*		*jacit*		*dsolve*	
Matrix	C2	Ci7	C2	Ci7	C2	Ci7	C2	Ci7	C2	Ci7
lung1	42.1	51.8	113.9	58.3	388.5	31.5	98.8	55.4	N/A	N/A
bcsstm34	24.3	6.2	53.6	29.5	22.7	5.6	27.2	6.7	19.8	4.0
cage9	21.0	8.1	44.3	26.1	22.1	8.1	28.6	10.3	2.9	2.0
rdist3a	17.9	5.6	39.5	21.1	17.7	5.2	-	-	3.2	2.1
jan99jac040	16.0	8.0	16.3	15.3	17.8	8.2	-	-	1.1	1.3
crystm01	8.3	4.9	17.1	17.0	9.1	4.9	10.8	5.8	2.2	1.8
ASIC_100ks	2.3	3.9	4.4	5.0	4.0	4.1	2.4	4.4	-	-
heart3	2.4	1.7	4.6	4.8	2.4	1.5	-	-	3.1	2.2
Zd_Jac3_db	2.5	1.6	4.6	4.8	2.6	1.7	2.6	1.9	-	-
Pres_Poisson	2.6	1.7	4.7	5.0	2.6	1.7	2.7	2.0	3.8	3.0
G2_circuit	2.6	4.7	4.6	4.7	5.0	5.1	2.6	5.7	-	-
bcsstk36	3.0	1.7	5.1	5.0	3.1	1.8	3.1	2.0	-	-
nd3k	3.5	1.9	5.4	5.2	3.5	1.9	3.6	2.1	-	-

Table 1. Number of iterations for the break-even points when tracing and restructuring is enabled, when using an initial random data layout. The matrices are ordered by increasing size. The lower part of the table contains the larger data sets, which do not fit in the caches. DSOLVE performs worse using *lung1* therefore a break-even point is not applicable. The missing entries for JACIT are due to zero elements on the diagonal. For DSOLVE the missing entries are due to matrices that take too long to factorize.

eration, the applications switches automatically to the non-traced version, which uses the restructured data. Four different matrices have been used which are representative in terms of performance characteristics (see Figure 4(a) and 4(b)). The break-even points for for all matrices are included in Table 1.

The figures show that tracing does come with an additional cost, but for most (larger) data sets the break-even point is reached within only a few iterations. For instance, for all data sets shown in Figure 6, the break-even point is reached within 4 iterations, except for *cage9*, which is the smallest data set depicted. Interestingly, on the Core i7, the break-even point is reached even quicker, making restructuring more attractive on this architecture.

4 Related Work

Optimization of data access in order to improve performance of data-intensive applications has been applied extensively, either by automatic transformations or by hand tuning applications for efficient access. In some cases, memory access patterns can be determined symbolically at compile-time and in such cases, the traditional transformations such as loop unrolling, loop fusion or -fission and loop tiling can be applied. For applications using pointer-linked data structures, such techniques can in general not be applied.

The methods above change the order of instruction execution such that data is accessed in a different way, without affecting the result. One might as well

change the underlying data layout, without affecting the computations. This is exactly what has been done on pointer-linked data structures in this paper.

In order to be able to automatically control the layout within type-unsafe languages such as C, a type-safe subset must be determined. The Data Structure Analysis (DSA) developed by Lattner and Adve does exactly that [6, 8]. It determines how data structures are used within an application. This has been discussed in Section 2.1.

DSA should not be confused with shape analysis. Shape analysis concerns the shape (e.g. tree, DAG or cyclic graph) of pointer-linked data structures. Ghiya and Hendren proposed a pointer analysis that classifies heap directed pointers as a tree, a DAG or a cyclic graph. Hwang and Saltz realized that it is of more importance how data structures are actually traversed instead of knowing the exact layout of a data structure. They integrated this idea in what they call *traversal-pattern-sensitive* shape analysis [4]. Integrating such an approach in our compiler could help in reducing the overhead introduced by the pool access tracing by traversing data structures autonomously in the run-time.

Type-safety is essential for data restructuring techniques. Two other transformations that use information provided by the DSA are structure splitting and pointer compression. Curial et al. implemented structure splitting in the IBM XL compiler, based on the analysis information provided by the DSA. Hagog and Tice have implemented a similar method in GCC [3]. The GCC based implementation does not seem to provide the same information as DSA. Strictly taken, structure splitting is not necessary for dynamic remapping of pointer structures, but it simplifies tasks like restructuring and relocation considerably. Moreover, splitting simply has performance benefits as data from unused fields will not pollute the cache.

Data layout optimization can also be provided by libraries. Bender and Hu proposed an *adaptive packed-memory array*, which is a sparse array that allows for efficient insertion and deletion of elements while preserving locality [1]. Rubin et al. take a similar approach by grouping adjacent linked list nodes such that they are colocated in the same cache line. They call this approach *virtual cache lines* (VCL) [10]. In their abstract, they state that they believe that compilers will be able to generate VCL-based code. We believe our pool restructuring does achieve this automatic remapping on cache lines. In addition, as our implementation employs full structure splitting, cache usage is very efficient after restructuring a memory pool.

Rus et al. implemented their Hybrid Analysis which integrates static and run-time analysis of memory references [11]. Eventually, such an approach might be useful in conjunction with our restructuring framework to describe access patterns of pointer traversals. Saltz et al. describe the run-time parallelization and scheduling of loops, which is an inspector/executor approach [12]. Our tracing mechanism is similar to this approach, as it inspects and then restructures. The future challenge will be to extend the system such that it inspects, restructures and parallelizes.

5 Conclusions and Future Work

In this paper, we presented and evaluated our restructuring compiler transformation chain for pointer-linked data structures in type-unsafe languages. Our transformation chain relies on run-time restructuring using run-time trace information, and we have shown that the potential gains of restructuring access to pointer-based data structures can be substantial.

Curial et al. mention that relying on traces for analysis is not acceptable for commercial compilers [2]. For static analysis, this may often be true. For dynamic analysis, relying on tracing is not necessarily undesirable and we have shown that the overhead incurred by the tracing and restructuring of pointer-linked data structures is usually compensated for within a reasonable amount of time, if data structures are used repetitively.

The restructuring framework as described in this paper opens up more optimization opportunities that we have not explored yet. For example, after data restructuring extra information on the data layout is available which could be exploited in order to apply techniques such as vectorization on code using pointer-linked data structures. This is a subject of future research.

Data structures that are stored on the heap contain object identifiers instead of full pointers. This makes the representation position independent, which provides new means to distribute data structures over disjoint memory spaces. Translation to full pointers would then be dependent on the memory pool location and the architecture. This position independence using object identifiers has been mentioned before by Lattner and Adve in the context of pointer compression [9]. However, with the pool restructuring presented in this paper, a more detailed segmentation of the pools can be made and restructuring could be extended to a distributed pool restructuring framework.

The implementation presented in this paper uses some run-time support functions to remap access to the proper locations for split pools. The use of object identifiers implies a translation step upon each load and store to the heap. These run-time functions are efficiently inlined by the LLVM compiler and have a negligible effect when applications are bound by the memory system. The run-time support could in principle be implemented in hardware and this would reduce the run-time overhead considerably. We envision an implementation in which pools and their layout are exposed to the processor, such that address calculations can be performed transparently. Memory pools could then be treated similarly to virtual memory in which the processors also takes care of address calculations.

We believe the restructuring transformations for pointer-linked data restructures that have been described in this paper do not only enable data layout remapping, but also provide the basis for new techniques to enable parallelizing transformations on such data structures.

References

1. Michael A. Bender and Haodong Hu. An adaptive packed-memory array. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 20–29, New York, NY, USA, 2006. ACM Press.
2. Stephen Curial, Peng Zhao, Jose Nelson Amaral, Yaoqing Gao, Shimin Cui, Raul Silvera, and Roch Archambault. Mpads: memory-pooling-assisted data splitting. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 101–110, New York, NY, USA, 2008. ACM.
3. Mostafa Hagog and Caroline Tice. Cache aware data layout reorganization optimization in GCC. In *Proceedings of the GCC Developers' Summit*, pages 69–92, 2005.
4. Yuan-Shin Hwang and Joel H. Saltz. Identifying def/use information of statements that construct and traverse dynamic recursive data structures. In *LCPC '97: Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*, pages 131–145, London, UK, 1998. Springer-Verlag.
5. Chris Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, May 2005. *See* `http://llvm.cs.uiuc.edu`.
6. Chris Lattner and Vikram Adve. Automatic pool allocation for disjoint data structures. *SIGPLAN Not.*, 38(2 supplement):13–24, 2003.
7. Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.
8. Chris Lattner and Vikram Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. *SIGPLAN Not.*, 40(6):129–142, 2005.
9. Chris Lattner and Vikram S. Adve. Transparent pointer compression for linked data structures. In *MSP '05: Proceedings of the 2005 workshop on Memory system performance*, pages 24–35, New York, NY, USA, 2005. ACM.
10. Shai Rubin, David Bernstein, and Michael Rodeh. Virtual cache line: A new technique to improve cache exploitation for recursive data structures. In *CC '99: Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, pages 259–273, London, UK, 1999. Springer-Verlag.
11. Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. *Int. J. Parallel Program.*, 31(4):251–283, 2003.
12. Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.*, 40(5):603–612, 1991.
13. Harmen L.A. van der Spek, Erwin M. Bakker, and Harry A.G. Wijshoff. Characterizing the performance penalties induced by irregular code using pointer structures and indirection arrays on the Intel Core 2 architecture. In *CF '09: Proceedings of the 6th ACM conference on Computing frontiers*, pages 221–224, New York, NY, USA, 2009. ACM.