Int J Parallel Prog (2008) 36:592–623
DOI 10.1007/s10766-008-0085-2

A Compile/Run-time Environment for the Automatic Transformation of Linked List Data Structures

H. L. A. van der Spek · S. Groot · E. M. Bakker · H. A. G. Wijshoff

Received: 6 June 2008 / Accepted: 19 August 2008 / Published online: 24 September 2008

Abstract Irregular access patterns are a major problem for today's optimizing compilers. In this paper, a novel approach will be presented that enables transformations that were designed for regular loop structures to be applied to linked list data structures. This is achieved by linearizing access to a linked list, after which further data restructuring can be performed. Two subsequent optimization paths will be considered: *annihilation* and *sublimation*, which are driven by the occurring regular and irregular access patterns in the applications. These intermediate codes are amenable to traditional compiler optimizations targeting regular loops. In the case of sublimation, a run-time step is involved which takes the access pattern into account and thus generates a data instance specific optimized code. Both approaches are applied to a sparse matrix multiplication algorithm and an iterative solver: preconditioned conjugate gradient. The resulting transformed code is evaluated using the major compilers for the x86 platform, GCC and the Intel C compiler.

Keywords Optimizing compilers · Parallel processing · Linked list data structures

1 Introduction

Recently, the emergence of multi and many-core architectures has raised a renewed interest in parallelizing compilers. In the past, most research has been conducted on the optimization of array-based codes, which are relatively easy to analyze. However, this certainly does not hold for computations using pointer structures. Pointers are often used to represent dynamic structures. Their advantage is that they can store these dynamic structures in a compact way. The disadvantage is that they prevent in-depth

H. L. A. van der Spek (✉) · S. Groot · E. M. Bakker · H. A. G. Wijshoff
LIACS, Leiden University, Leiden, The Netherlands
e-mail: hvdspek@liacs.nl

dependence analysis. As a consequence, the majority of the compilers in use today will fail in optimizing loop structures that contain pointer traversals.

For instance, one very common pointer structure is the linked list. A linked list represents a sequence of elements in which each element is in a potentially unrelated memory location, making it impossible for the compiler to optimize the memory access patterns. The presence of the code that accesses the linked list impedes analysis of the code (e.g., loop-carried data dependence analysis) and thus prevents the application of optimizing transformations such as loop interchange or even more drastic code restructuring. The irregular access pattern also poses a problem for the CPU cache, which cannot exploit the locality of subsequent memory references. Also computations performed on subsequent items cannot be vectorized, if a linked list is used.

To overcome these limitations, we propose a sequence of transformations that will allow specific pointer chain traversals to be transformed into regularly accessed array-based codes. Such a representation is more suitable for analysis and allows the application of many of the optimizations which were originally designed for regular arrays. Moreover, the transformation framework described here intends to be generically applicable to a wide range of loop structures that use linked lists, i.e., loop structures that iterate over linked lists. The resulting intermediate form is further optimized using two different restructuring strategies, *annihilation* and *sublimation*, both of which are driven by array access patterns.

To illustrate the intermediate form and the two restructuring techniques we give a simple example, which performs sparse matrix times vector multiplication (Ab). The rows are stored using linked lists, which are traversed during the multiplication:

```
for(i = 0; i < m; i++) {
    p = Matrix->Row[i];
    while(p) {
        x[i] = x[i] + p->Value * b[p->ColIndex];
        p = p->next;
    }
}
```

This linked list based implementation of matrix multiplication would then be transformed into the following intermediate code:

```
for(i = 0; i < m; i++) {
    p = Matrix->Row[i];
    n = 0;
    while(p) {
        A[n] = p->Value;
        C[n] = p->ColIndex;
        p = p->next;
        n++;
    }
    for(j = 0; j < n; j++)
        x[i] = x[i] + A[j] * b[C[j]];
}
```

Note that while in this example the extra overhead is still within the outer *for*-loop, this can often be eliminated by hoisting the traversal code outside the outer loop. This is described in Sect. 3.8. The inner *for*-loop contains two access patterns, one of which is the pattern induced by the simple loop counter j and one which is induced by $C[j]$. For restructuring using annihilation, array B is restructured to follow the regular access pattern j of A. This would lead to the following code:

```
for(i = 0; i < m; i++) {
    p = Matrix->Row[i];
    n = 0;
    while(p) {
        A[n] = p->Value;
        C[n] = p->ColIndex;
        p = p->next;
        n++;
    }
    for(k = 0; k < n; k++)
        B'[k] = b[C[k]];
    for(j = 0; j < n; j++)
        x[i] = x[i] +  A[j] * B'[j];
}
```

In the other case, when sublimation is applied, A is restructured to follow the irregular access pattern induced by $C[j]$, under the assumption that this access pattern is injective (has no duplicate values). This results in the following code:

```
for(i = 0; i < m; i++) {
    p = Matrix->Row[i];
    n = 0;
    while(p) {
        A[n] = p->Value;
        C[n] = p->ColIndex;
        p = p->next;
        n++;
    }
    for(k = 0; k < n; k++)
        A'[C[k]] = A[k];
    for(j = 0; j < n; j++)
        x[i] = x[i] + A'[C[j]] * b[C[j]];
}
```

Note that the irregularity is still present in the inner loop. However, the access patterns of both accessed arrays are identical in this case. In Sect. 3.7, it is shown how these loops are transformed into a fully regular intermediate code, which thereupon can be used for analysis.

Figure 1 shows the architecture of our restructuring framework. In the first step, the C code is converted to a normalized representation. The rest of the compile-time part of the system concerns the identification of list traversals and the subsequent

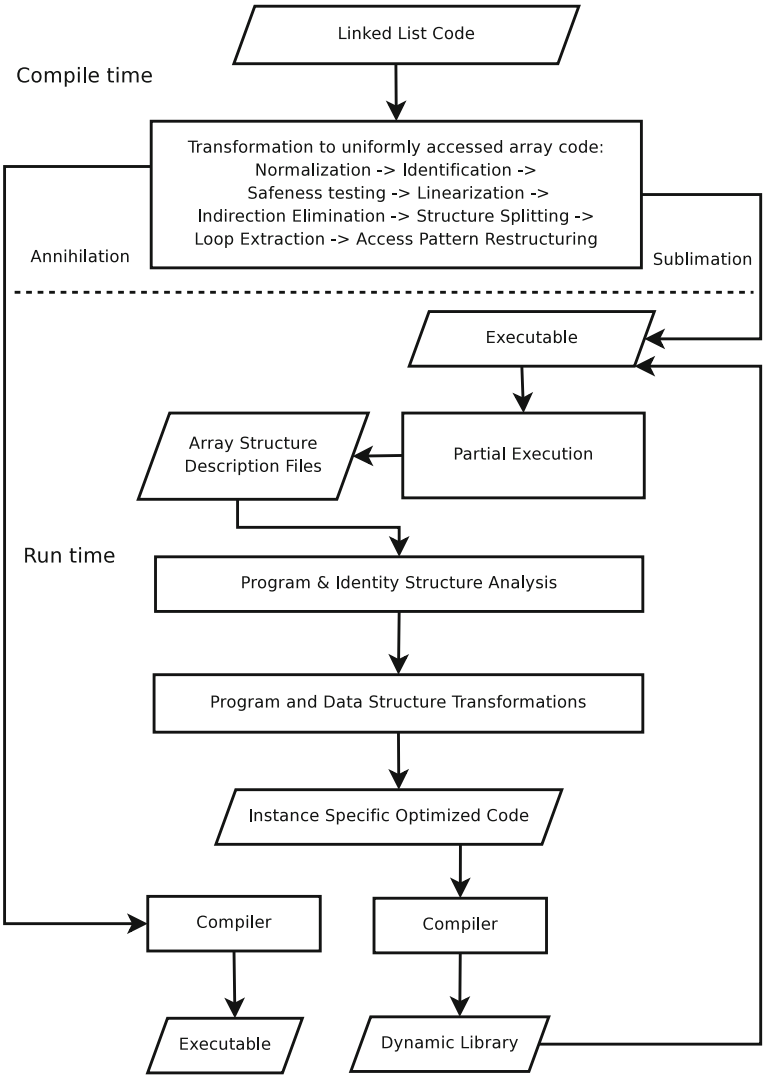

Fig. 1 Linked list restructuring compiler architecture

transformations to obtain an array-based intermediate code. These steps are explained in Sect. 3.1 through 3.8. In the case of annihilation, the resulting code is directly executable. For sublimation, the resulting intermediate must be recompiled at run-time. The reason for this is that when applying annihilation, the exact index sequence is known at compile time ($0 \leq i \leq n$, stride 1), but for sublimation, the content of the index array is only known at run-time. This is explained in more detail in Sect. 3.6. For sublimation, the resulting executable will partially execute the code, until the point where the index patterns are known. Then the intermediate code is recompiled, together with information from the index arrays. Section 3.9 describes this mechanism.

In this paper, the different steps of the transformation framework will be illustrated by their application to a code example which performs matrix multiplication (Sect. 4). We also perform experiments with this code and an additional code performing Preconditioned Conjugate Gradient which both have been transformed by our prototype compiler. The results of these experiments are discussed in Sect. 5. Finally, we will discuss future directions. A brief description of this work has been published by Groot et al. [15].

2 Related Work

Much work has been done in optimizing code to reduce latency caused by cache-misses. Our work follows a different philosophy: the regular intermediate code that is generated is suitable for dependence analysis. This enables a much richer set of further optimizations, such as optimizations for the memory hierarchy, but also loop parallelization. While our paper does not focus on alias and shape analysis, such methods are required to detect which traversals can be safely transformed.

One way of obtaining shape analysis relies on the structural properties of specific recursive data structures, which can enable a wider range of optimizations. Ghiya and Hendren proposed a pointer analysis which classifies heap directed pointers as either a *tree*, a *DAG* or a *cyclic graph* [11]. This information on disjointness and cyclicity is valuable when selecting a pointer traversal to analyze. However, as Hwang and Saltz point out, it is not only important what the actual structure *is* [18], but in addition, how it is actually *traversed*. They call this traversal-pattern-sensitive shape analysis. Lattner and Adve implemented their Data Structure Analysis (DSA), which identifies disjoint data structures [22]. DSA provides information about type-safety and connectivity information which aides in the selection of suitable traversals. However, these approaches are not alternatives to our approach, they rather are analysis tools which can be used to assess the safety of the transformations presented in this paper. Analysis techniques like those described here will be incorporated into a future implementation of our framework.

Bender and Hu [1] describe an *adaptive packed-memory array*. This is a sparse array structure which allows for efficient insertion and deletion of elements while preserving locality. A similar approach is taken by Rubin et al. [27], who describe a data structure that maintains groups of adjacent linked list nodes in such a way that they are loaded into a cache line simultaneously. This concept is called *virtual cache lines*. Both these approaches require a programmer to choose these specific data structures. In our approach the compiler automatically decides on these data structures based on the code and data on which the code operates.

Prefetching is a technique that is more easily employed by a compiler. By issuing a prefetch instruction before other work is done in a loop, memory latency caused by a cache-miss on a pointer traversal can be hidden, provided that the work done in the loop body is substantial. Karlsson et al. [19] describe techniques to prefetch irregular accessed linked structures which extends the work of Luk and Mowry [24], who mention *data linearization* as a technique to improve prefetching efficiency and locality. Yang and Lebeck [33] present a memory architecture which pro-actively dereferences

pointers lower in the hierarchy to push data higher into the memory hierarchy instead of waiting for the higher memory-level to pull the data. While these approaches do improve performance, the linked list pattern remains in the loops which impedes dependence analysis and hence optimizations like concurrentization and vectorization as described by Padua and Wolfe [26] cannot often not be applied.

Other techniques that can be applied are *structure splitting* as proposed by Chilimbi et al. [6], and Zhong et al. [35], who also describe the opposite transformation *array regrouping*. *Field reordering* as described by Chilimbi et al. [6] is another method to increase locality in recursive data structures. Structure splitting is a transformation in which so called hot and cold regions are identified after which the structure is split accordingly, which improves cache behavior. Array regrouping merges different arrays into a new array of structures, which can increase locality of reference. Field reordering clusters structure members such that their ordering matches the spatial access pattern, which may be quite different from the definition given by the programmer. Techniques like these are finding their way in production compilers. Hagog and Tice [16] have described an implementation of structure splitting and structure reordering techniques into the GCC compiler. Golovanevsky and Zaks [13] have described further progress on the implementation.

Approaches which are related to our research are solely based on symbolic and compile-time analysis to improve the performance of pointer-structured computations. Examples are the work of Chong and Rugina [7] and Rugina and Rinard [28], in which access pattern analysis is performed symbolically by the compiler to obtain bounds on memory access regions.

Most optimizations are performed at compile-time. As program behavior is dynamic, run-time optimizations can result in more efficient code, because more complete information is available. However, if static analysis can be done this is preferable. Rus et al. [29] describe a framework which integrates static and dynamic memory reference analysis. This is done by generating code which will finish analysis at run-time for cases that cannot be statically determined. Saltz et al. [30] use an *inspector loop* at run-time to determine wave-fronts of concurrently executable loop iterations. This bears some resemblance to our approach, however, our data specific memory access pattern restructuring is a different concept. Work by Lin and Padua [23] involves the analysis of injectiveness of indirect addressed accesses to exploit the automatic parallelization of irregular, indirect addressed loop structures. As such, it is different from the work described in this paper, although it could be seen as a very limited application of the techniques we propose. A data-centric approach is proposed by Kodukula and Pingali [20], in which the compiler decides in which order data elements are brought into the cache. The order in which statements are executed is therefore governed by the order in which data is fetched. This method is only applicable to dense FORTRAN-like codes.

The approach as described in this paper is essentially different from the other approaches discussed above. By linearizing access to a linked list, linked list iteration statements are transformed into simple *for*-loops of which the access patterns can be restructured by applying techniques similar to those described by Zhao and Wijshoff [34]. This leads to an intermediate code that is amenable to further optimizations which take into account the characteristics of the underlying data structures. In the case of

sublimation, a partial execution is required followed by a run-time recompilation of part of the code, resulting in highly efficient code that is specialized for a particular instance of a data structure.

3 Transformation Steps

This section describes the steps a compiler should take to transform code using a linked list access pattern into code that is optimized for a specific instance of a linked list. A rough outline of these steps can be found in Fig. 1. The compile-time step as depicted should be seen as consisting of multiple transformation steps which were grouped together because they strongly interact.

3.1 Normalization

Programming languages often contain complex expressions. In order to make transformations easier, a normalization step is performed before all other steps. In this paper, all transformations are source-to-source transformations written for C. The normalization step involves the conversion of *for*-loops to *while*-loops, expression flattening, and common subexpression elimination. These steps lead to a form of C code that is easier to transform.

The loop normalization results in a uniform representation for list traversals. The following rule is used:

```
for(init; cond; iter) {block;} ->
     {init; while(cond) {block; iter;}}
```

Next, all code is *flattened*. In this form, complicated expressions are unraveled and their results are stored in temporary variables. Also pointer expressions that are dereferenced are first put into a temporary variable. This process can be viewed as creating a type of three address code for C. It drastically reduces the number of cases to consider and enables transformations on, for instance, an array of linked lists.

The last step in the normalization phase is common subexpression elimination which together with the flattening of expressions results in easily identifiable linked list traversals. An example where this transformation enables simple recognition of a linked list traversal is the following code:

```
/* list[x] = list[x]->Next; */
*(list + x) = (**(list + x)).Next;
```

Flattening all expressions results in:

```
temp1 = lists + x;
temp2 = lists + x;
*temp1 = (*(*temp2)).Next;
```

After common subexpression elimination, code in which a linked list traversal has a simple representation results:

```
temp1 = lists + x;
*temp1 = (*(*temp1)).Next;
```

(Note that in the examples used in this paper, non-normalized code is used to keep the examples more readable).

3.2 Identification of Linked List Traversals

Linked list traversals can be split in two parts, a part that performs the computations and a part that navigates through the linked list, as pointed out by Ghiya et al. [12]. The first step in identifying such loops consists of identification of possible linked list candidates, which are structures containing members pointing to the same data structure as the containing data structure (recursive data structure). The following structure declaration illustrates the general pattern of a recursive data structure:

```
struct datatype {
    double data;
    ...
    struct datatype *Next; /* Candidate */
    ...
};
```

Recursive data structures are often traversed using the following pattern:

```
...
node = begin_list;
...
while(node != end) {
    /* Some computations */
    node = node->Next;
}
```

Ghiya et al. [12] describe a method for identifying linked list traversals which do not have loop carried dependences (apart from the iteration pointer itself). In this section, the properties of this method that are essential for the safeness of the transformations proposed are briefly reviewed.

The only exit point from the loop should be the loop condition, i.e., statements like *continue*, *break*, *goto* and *return* are not allowed. Function calls which cannot be analyzed (e.g., calls to libraries) are forbidden as well, as their side effects cannot be determined.

The condition of the *while*-loop tests whether the end of the list has been reached. An iteration pointer (called *navigator* by Ghiya et al. [12]) is identified by computing definition chains (only using loop-resident definitions) for the loop condition. A variable is considered a navigator, if it is uniquely defined by one loop-resident statement and if this statement contains a recurrent definition of that variable. Moreover, this statement must be executed unconditionally, i.e., it must appear at the top level of the *while*-loop body. Any other variable in the loop condition must be loop-invariant, i.e., it should have no loop-resident definition.

Additional checking of the navigator is necessary to guarantee that a node is not revisited during the traversal. This means that there may not be any modifications of the fields that are used to traverse the list. In order to prove safety of the transformations, pointer analysis techniques must be used. The work of Ghiya [10] shows an extensive overview of pointer analysis techniques in the context of recursive data structures.

Many linked list traversals can be handled in this manner. Another assumption that must be made is that the *while* loop under consideration will terminate, which is a reasonable assumption as algorithms that traverse structures should terminate. Note that even cyclic linked lists can be handled, as the loop termination condition together with the loop analysis guarantees that no elements will be revisited. Under these conditions, cyclic lists do not require special attention.

3.3 Linearization

Linearization is the process of traversing a linked list and storing the pointers that are encountered during this traversal. The original iteration loop can then be replaced with a *for*-loop iterating over the newly created array. All linked list pointers are then replaced with an array reference. Applying linearization on the loop structure from Fig. 2 results in the following code (memory allocation/deallocation code is omitted):

```
i = 0;
/* Pre-initialization: linearize linked list */
while(node != end) {
  A[i] = node;
  node = node->Next;
  i++;
}
iMax = i;
/* Substitute linearized array for
   iteration pointer */
for(i = 0; i < iMax; i++) {
  ... = A[i]->Value * B[idx_expr];
}
```

The loop in which the list is linearized is called the *pre-initialization loop*.

3.4 Indirection Elimination

The linearization step produces an array with pointers to linked list elements. Using these pointers, data members are accessed and used in the computation loop. This

Fig. 2 Structure of a loop using a linked list and array *B*

```
while( node != end ) {
   ... = node->Value *  B[idx_expr];
  node = node->Next;
}
```

indirection can be removed by changing the pre-initialization loop and performing the indirection there. Thus, the pre-initialization loop will acquire an extra level of indirection, but in the computation loop, one level of indirection is eliminated. The code generated by the linearization step would be transformed as follows:

```
i = 0;
while(node != end) {
  A[i] = *node; /* Do indirection here */
  node = node->Next;
  i++;
}
iMax = i;
for(i = 0; i < iMax; i++) {
 /* Result: indirection is eliminated here */
  ... = A[i].Value * B[idx_expr];
}
```

3.5 Structure Splitting

It is inefficient to copy the entire structure in the initialization loop. Many structure members may not be used in the computation loop and such data would unnecessarily reside in the cache. Additionally, a non-unit stride access pattern on the arrays can prevent other optimizations such as vectorization. By only copying the members which are actually needed, these problems are circumvented. It also leads to code that is easier to analyze. Applied to the code resulting from the indirection elimination step, the following code is obtained:

```
i = 0;
while(node != end) {
  A_Value[i] = (*node).Value;
  node = node->Next;
  i++;
}
iMax = i;
for(i = 0; i < iMax; i++) {
  ... = A_Value[i] * B[idx_expr];
}
```

3.6 Access Pattern Restructuring

The newly generated computation loop contains an array which is indexed by a new iteration variable (i in this example). Other arrays within the same loop do not follow the access pattern induced by this variable, although the access pattern may be dependent on the linearized linked list. Consider the computation loop obtained in the previous step. For readability, A_Value is renamed to A.

```
for(i = 0; i < iMax; i++) {
  ... = A[i] * B[idx_expr];
}
```

This example contains two different access patterns, namely

1. the access pattern induced by i (such a pattern always exists after linearization) and
2. the access pattern induced by idx_expr.

In order to impose the same access pattern onto both arrays, either A must be accessed using the access pattern of B or B must be accessed using the access pattern of A. Note that the index expression of B may originally have been dependent on the linked list. This expression will have been converted to an array by the indirection elimination and structure splitting step: e.g., $B[A[i] \rightarrow Index]$ would have been transformed to $B[A\_Index[i]]$.

If the array B is remapped using the index expression idx_expr, idx_expr must be injective (the index expression is viewed as a function of i), otherwise at least one element becomes inaccessible. For example, consider the access patterns (1, 2) for A and (1, 1) for B. The access pattern of B is not injective and as $A[1]$ can contain only one value, the original semantics of the loop cannot be reproduced.

Consider the two access pattern restructuring cases, 1 and 2, *annihilation* and *sublimation*, respectively. It is assumed that A is indexed using iteration counter i, with lower and upper loop bounds $iMin$ (=0) and $iMax$, respectively. B is indexed by idx_expr, which is an injective index expression dependent on i.

1. (*Annihilation*) Impose the access pattern of A onto B, that is, restructure based on the index expression of A, which is i. This restructuring is done by creating a new array B' which is defined as follows:

$$B'[i] = B[idx\_expr],\ \forall i(iMin \leq i \leq iMax)$$

   This case is very intuitive: fetch the elements actually needed from the other array B and rewrite the loop such that the restructured array is tightly packed and accessed in the same order as A. Note that changing the access pattern of B to follow the access pattern of A is always possible if B is not written to, since the access pattern induced by i is injective.
2. (*Sublimation*) Impose the (irregular) access pattern of B onto A, that is, restructure based on the index expression of B, which is idx_expr. This restructuring is done by creating a new array A' which is defined as follows:

$$A'[j] = \begin{cases} A[i] & \text{if } \exists i(idx\_expr(i) = j) \\ identity & \text{otherwise} \end{cases}$$

   This case is not very intuitive: the variable j ranges from $-\infty$ to ∞, and therefore this definition is only semantic. At run-time, the compiler will be able to extract proper loop bounds from this definition. The identity value depends on the computation in which the data is used. This is explained in the next section.

3.7 Iteration Space Expansion

In case of sublimation, not every element of B (see previous section) is necessarily accessed. For instance, if throughout the execution of the loop the variable idx_expr defines the sequence (5, 50, 10), then only the elements $A'[5]$, $A'[50]$ and $A'[10]$ need to be defined.

However, if the new iteration space is defined to be the interval $[-\infty, \infty]$, the other elements of A' must not alter the semantics of the program. Therefore, the "gaps" in A' should be filled with values chosen in such a way that they do not have any effect when the code is executed. For example, when the loop executes a statement like

$$X = X + A'[i] * Y,$$

then the so called *identity* value must be 0, as this will preserve the semantics of the program. The resulting code will have infeasible loop bounds, but this code only serves as an *intermediate*. The advantage of this approach is that information about the sparse structure is preserved and together with input data, a compiler can determine if other access functions can be used, for example storing diagonals of a matrix separately. This approach has already been proven successfully on both dense FORTRAN codes (Bik et al. [3]) and sparse codes (Zhao and Wijshoff [34]).

3.8 Loop Extraction

The pre-initialization loop that is generated is placed into the same compound statement as the original linked list traversal. Therefore, the new loop could end up nested within other loops. In order for the transformations to be efficient, the initialization loop should be extracted from this loop. This transformation enables further optimizations as it often results in a perfectly nested loop. If loop extraction cannot be performed, due to dependences, another access pattern should be tried in the restructuring phase. This section describes how loop extraction should be performed.

The pre-initialization loop uses a number of variables. Some of these are generated replacement variables which will be used within the transformed main loop. Other variables are used for bookkeeping the pre-initialization loop, such as variables tracking loop boundaries. All remaining variables are non-generated variables or expressions originating from the initial code.

If an initialization loop A is contained in an outer loop O (Fig. 3), it can be extracted using one of the following two techniques:

1. If all non-generated variables used in A are invariant over O, then A is just repeating the same operation every iteration of O. It is therefore safe to move A in front of O without any further processing.
2. If some non-generated variables used in A are not invariant over O, but all non-invariant variables are only dependent on a loop counter with known bounds, then A can be extracted from O.
   This requires extension of the generated variables, which is done by adding an extra dimension to these variables, such that for every iteration the correct value is

Before:

```
loop O {
  loop A {
    generated-variable =
        original-variable;
   }
   /* additional code including
      transformed main loop
      referencing generated-variable */
}
```

After:

```
loop O' {
  loop A {
    generated-variable[E] =
        original-variable;
   }
}
loop O {
   /* additional code including
      transformed main loop
      referencing generated-variable[E] */
}
```

Fig. 3 Loop A which depends on E is extracted from O

preserved. All references to generated variables must be changed to use the new, extended variables. A new loop O' can be created which uses the original loop structure of O. Subsequently, loop A is moved from O to O'.

Loop extraction plays a vital role in the transformation chain. If any dependence prevents loop extraction, the restructuring code and computation code are not separated. In the case of annihilation, this results in code where the restructuring is done every iteration, which in itself is expensive and prevents the computation loops to become perfectly nested (which in turn prevents further optimizations). In the case of sublimation, the lack of separation of restructuring code and computation code prevents recompilation and therefore this optimization path is disabled. An example of a loop containing such a dependence is an algorithm for Preconditioned Conjugate Gradient (PCG). Golub and Van Loan [14] provide a clear description of this algorithm in their book. The algorithm, which is not discussed here in detail, contains a sparse matrix times dense vector multiplication that is embedded in a loop. It has the following structure:

```
while(...) {
  ...
  /* Some definition of vector p */
  ...
```

```
    for(row = 0; row < rows; row++) {
      /* Linearization code omitted */
      /* Restructuring code */
      for(i = 0; i < iMax; i++)
        p'[i] = p[A_ColIndex[i]];

      for(i = 0; i < iMax; i++)
        result[row] += A_Value[i] * p'[i];
    }
    ...
  }
```

The redefinition of p makes the application of annihilation less attractive, because the restructuring code cannot be moved in front of the containing *while*-loop. As an alternative, sublimation can be applied (using the access pattern $A\_ColIndex[i]$). In Sect. 5.2, a comparison of annihilation and sublimation is made using PCG based on linked lists.

3.9 Run-time support

In the case that sublimation is applied, the code obtained from the previous steps (which all are performed at compile-time) results in a code that is never executed. A partial recompilation step is included in the run-time system where the newly generated loops are transformed. Only the computation loops are recompiled while keeping the pre-initialization untouched. During execution of the pre-initialization loop, the non-identity structure of the restructured array is written to a file which is used in the recompilation step to generate instance specific optimized code.

The perfectly nested loop is recompiled using the additional non-identity structure information. The implementation of such a compiler is discussed in depth by Bik and Wijshoff [4, 5], and Bik [2]. This recompilation step emits restructured code which is compiled into a shared library which in turn is loaded by the application at run-time. Finally, the newly generated code is executed.

4 Example

The concepts described is this paper is demonstrated by using a code example of sparse matrix multiplication. Figure 4 depicts the data structure used for the representation of a sparse matrix. Figure 5 shows the actual C code performing the multiplication. Only the left matrix is sparse (compressed row storage), the right matrix and the result matrix are both dense. The rows of the sparse matrix are traversed using linked lists which prevents optimizations such as loop interchange and vectorization. Additionally, as linked list elements cannot be assumed to be successive elements in main memory, performance will suffer due to cache misses.

If we consider the loops in the example code, one of these loops, the *while*-loop, is a traversal of a linked list. With an $n \times m$ result matrix, this loop is executed $n \cdot m$

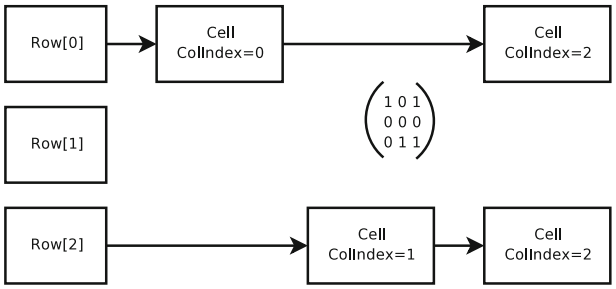

Fig. 4 A sparse matrix using a linked list representation

Fig. 5 Sparse matrix multiplication

```
void MatrixMultiply(Matrix left,
        double **right, double **result,
        int cols )
{
  Cell *leftCell;
  int dimensions = left.Dimensions;
  int row, col;
  for(col=0; col<cols; col++) {
    for(row=0; row<dimensions; row++) {
      leftCell = left.Rows[row];
      while( leftCell != NULL ) {
        result[row][col] +=
            leftCell->Value *
            right[leftCell->ColIndex][col];
        leftCell = leftCell->ColNext;
      }
    }
  }
}
```

times, because the dot product of each row of the left matrix with each column of the right matrix involves the traversal of a linked list which is relatively costly. Ideally, the inner loop would be vectorized, however this is prevented by using the contents of a linked list element as operand for the multiplication ($leftCell \rightarrow Value$) and for indexing of an array ($leftCell \rightarrow ColIndex$).

This code can be transformed to an intermediate code which is equivalent, but uses directly accessed arrays in the inner loop. As an initial step, linearization is applied on the linked list which transforms the linked list traversal into a pre-initialization loop and a computation loop.

On the resulting code, indirection elimination and structure splitting can be applied. In the computation loop, only the members $Value$ and $ColNext$ are used. These members will be put into an array by applying structure splitting. The code produced by these steps can be found in Fig. 6. Not all intermediate steps are shown in detail. Memory management code is inserted to dynamically resize arrays when needed at run-time. This is necessary, because the length of a linked list is not known a priori.

```
#define INIT_ALLOC 32
void MatrixMultiply(Matrix left, double **right,
                    double **result, int cols )
{
  Cell *leftCell;
  int dimensions = left.Dimensions;
  int row, col, i, j;
  double **A_Value = (double **)malloc(sizeof(double *) * dimensions );
  int **A_ColIndex = (int **)malloc(sizeof(int *) * dimensions );
  int counter;
  /***SEMANTIC_DEFINITION A_Valuep
    forall i,j {
      A_Valuep[i][j] = 0; }
  ***/
  for( row = 0; row < dimensions; row++ ) {
    int size = INIT_ALLOC;
    A_Value[row] = (double *)malloc(sizeof(double)*size);
    A_ColIndex[row] = (int *)malloc(sizeof(int)*size);
    leftCell = left.Rows[row];
    counter = 0;
    while( leftCell != NULL ) {
      if( counter == size ) {
        /* Resize array */
        size *= 2;
        A_Value[row] = (double *)realloc(A_Value,
                                         sizeof(double) * size );
        A_ColIndex[row] = (int *)realloc(A_ColIndex,
                                         sizeof(int)*size );
      }
      /* Code generated by successive application of linearization,
       * indirection elimination and loop extraction (which introduced
       * the extra dimension indexed by row) */
      A_Value[row][counter] = (*leftCell).Value;
      A_ColIndex[row][counter] = (*leftCell).ColIndex;
      leftCell = leftCell->ColNext;
      counter++;
    }
    /***RESTRUCTURE A_Value
    forall i ( 0 < i < counter ) {
      A_Valuep[row][A_ColIndex[i]] = A_Value[i]; }
    ***/
  }
  /* This loop will be recompiled and loaded dynamically */
  /***RECOMPILE
  for( col = 0; col < cols; col++ ) {
    for( row = 0; row < dimensions; row++ ) {
      forall j {
        result[row][col] += A_Valuep[row][j] * right[j][col];
      }
    }
  }
  ***/
  for( row = 0; row < dimensions; row++ ) {
    free( A_Value[row] ); free( A_ColIndex[row] );
  }
  free( A_Value ); free( A_ColIndex );
```

Fig. 6 Sparse matrix multiplication after loop extraction (sublimation)

At this point, two choices can be made: either the access pattern of A_Value is used to restructure $right$ (annihilation) or the access pattern of $right$ is used to restructure A_Value (sublimation). Memory allocation code is left out of these short code samples.

4.1 Annihilation

In the first case, $right$ is restructured as follows, following the definition from Sect. 3.6:

```
forall (0 <= i < n) {
  rightp[i] = right[A_ColIndex[i]]; }
```

This is the semantic definition of $rightp$. As annihilation can be done at compile-time, this definition is translated into C code. $rightp$ ($right$ prime) is now substituted for $right$ in the computation loop:

```
forall i (0 < i < counter) {
  result[row][col] += A_Value[i] * rightp[i][col]; }
```

4.2 Sublimation

In the second case, A_Value is restructured to follow the access pattern of $right$. This is done by defining A_Valuep as follows:

```
forall i {
  A_Valuep[i] = 0; }

forall i (0 < i < counter) {
  A_Valuep[A_ColIndex[i]] = A_Value[i]; }
```

Contrary to the restructuring code used by annihilation, the restructuring code emitted by sublimation is a semantic definition which *cannot* be compiled to executable code until the actual access pattern is known at run-time. Therefore, the definition of the restructuring code is never really executed. It only tells the compiler that an array element of A_Valuep that is not defined by the original array A_Value should be regarded as being 0. In principle, the compiler is free to materialize some of these elements at run-time, if this is beneficial (e.g., materialize an entire diagonal of a matrix, including some zero values). Of course, other computations might require a different identity value.

Now A_Valuep can be substituted for A_Value:

```
forall i (0 < i < counter) {
  result[row][col] += A_Valuep[A_ColIndex[i]] *
                      right[A_ColIndex[i]][col]; }
```

4.3 Iteration space expansion

In order to remove the indirect addressing from the loop, change the loop control structure such that the loop iterates over the entire iteration space that can be spanned by the iteration variables

```
forall j {
  result[row][col] += A_Valuep[j] * right[j][col]; }
```

Remember that this does not change the semantics of the program, as the "gaps" in A_Valuep have been set to 0. However, the iteration space is now unbounded, and this intermediate code cannot be executed. At run-time, this intermediate code is recompiled, using the additional indexing information. At that point, new loop structures are generated with appropriate loop bounds.

4.4 Loop Extraction

The pre-initialization code depends on the variable row, which is not loop invariant (the containing *for*-loop increases it by one every iteration). Therefore, loop extraction cannot be directly applied without eliminating this dependence. No value which is pointed to by *leftCell*, either directly or indirectly (by following a pointer chain), is modified. Therefore, loop extraction is possible if all variables dependent on $leftCell$ are extended, such that for every iteration of row, the values which were copied are stored separately. This is a form of data privatization.

The pre-initialization code is now independent of the computation loop and can be moved in front of all containing *for*-loops. The resulting code is shown in Fig. 6 which is code resulting from restructuring A_Value. The extraction of the pre-initialization code introduces a new dimension for some variables. The definition of the restructured variable A_Value is adapted accordingly. This code sample also includes memory-management code for the linearization and structure splitting steps.

An interesting issue is the determination of the loop bounds in the computation loops. Instead of having a separate bounds for each row (for which code could be generated), no bounds are specified at all, i.e., the entire iteration space is used. This extends the iteration space of the computation loop but poses no further problems, as the compiler can determine feasible loop bounds at run-time, using the fact that any element of A_Valuep that has not been assigned a value from A_Value has the identity value (in this case 0).

The resulting code only traverses the linked list once per row and the computation loop has become a perfectly nested *for*-loop, which is easier to analyze. Loop interchange is enabled in this case and subsequently, the inner loop can be vectorized. Additionally, this loop structure dramatically increases cache performance as subsequent items are adjacent in memory.

When applying annihilation, the run-time recompilation phase is left out, as the iteration space is defined (symbolically) at compile-time. As mentioned before, the version resulting from restructuring A_Value to match the access pattern of $right$ induced by $A\_ColIndex[i]$ (sublimation) is not directly executable, as this code

can have a (at least theoretically) infinite iteration space. Together with non-identity structure information the computation loop can be transformed to a data instance specific code which can be very efficient, as will be shown in the following section.

5 Experiments

5.1 Matrix Multiplication

The transformations described in this paper have been applied on code performing matrix multiplication, as described in the above example. In this case, the left matrix is a sparse $n \times n$-matrix and the right matrix is a dense $n \times m$-matrix. The result matrix is also dense. Figure 5 shows the original algorithm.

All benchmarks have been executed using both the GNU C/FORTRAN compiler and the Intel C/FORTRAN compiler. As the semantic definitions of the restructured arrays are language independent, the semantic definitions are translated to FORTRAN, for which a restructuring compiler called MT1 (Bik and Wijshoff [4, 5], and Bik [2]) exists that exploits the non-zero patterns of arrays given dense algorithm definitions. There are five different versions of the program:

- The original program.
- Two programs generated using MTC, our prototype compiler, which performs the compile-time part of sublimation. For the run-time recompilation, one version uses the FORTRAN compiler to compile the code emitted by MT1 and one program uses *f2c*, a FORTRAN to C compiler by Feldman et al. [9], to convert the code emitted by MT1 back to C. The output of *f2c* is further compiled by the C compiler.
- Two programs obtained using annihilation (one without further optimizations and one with loop interchange applied).

All programs have been compiled using both the GCC and the Intel Compiler.

For sublimation, the access pattern chosen is the access pattern of $right$, which leads to the following computation loop:

```
for(col = 0; col < cols; col++) {
  for(row = 0; row < dimensions; row++) {
    forall i {
      result[row][col] +=
          A_Valuep[row][i] * right[i][col];
    }
  }
}
```

The real bounds of the iteration space are obtained at run-time using the semantic definition of the restructured arrays. Again, note that the version obtained using sublimation is an intermediate code which should not be executed. In the example shown, a sparse matrix structure is embedded into a two dimensional array whose size in

principle is infinite. As explained in Sect. 3.6, A_Value is restructured to follow the access pattern of $right$. The $identity$ value for the multiplication followed by addition is 0. The access pattern of A_Valuep is determined in the pre-initialization loop and can be used to restructure the loops such that this structure is taken into account during optimization. Currently, the run-time restructuring step (sublimation) is done by transforming the semantic definitions to FORTRAN code for which such restructuring techniques have been implemented (MT1). The restructured code is compiled and linked to the calling application at run-time.

The other versions are generated by applying annihilation, i.e., $right$ is restructured to follow the access pattern of A_Value, which is i. This results in the following computation loop (defined as C code):

```
for(col = 0; col < cols; col++) {
  for(row = 0; row < dimensions; row++) {
    for(i = 0; i < iMax[row]; i++) {
      result[row][col] +=
          A_Value[row][i] * rightp[row][i][col];
    }
  }
}
```

After applying loop interchange:

```
for(row = 0; row < dimensions; row++) {
  for(i = 0; i < iMax[row]; i++) {
    for(col = 0; col < cols; col++) {
      result[row][col] +=
          A_Value[row][i] * rightp[row][i][col];
    }
  }
}
```

The compilers used are GCC 4.0.2 and the Intel C and FORTRAN Compiler 9.1. The benchmarks were executed on an Intel Pentium 4 Xeon 3.00 GHz CPU (family 15, model 4, stepping 1), 4GB of main memory running SuSE Linux 10.0. Four different matrices which are publicly available MatrixMarket [25]. were used, namely *sherman3*, *memplus*, *add32* and *af23560*. These originate from the Harwell-Boeing Matrix Collection which is described by Duff et al. [8].

Figure 7 shows the execution times for sparse matrix times dense matrix multiplication for various sizes of the right matrix using the GCC compiler. Figure 8 shows the results for the Intel Compiler. The speedups relative to the original code (GCC code relative to the original code compiled with GCC, Intel Compiler code relative to the original code compiled with the Intel C compiler) when the right matrix has 3000 columns, is shown in Fig. 9.

As can be seen in Figs. 7 and 8, the run-time recompilation can take a considerable fraction of the total execution time, but it dramatically speeds up the computation itself. In this experiment, the code generated by applying annihilation clearly is the fastest. Table 1 shows the initialization time needed by annihilation and sublimation.

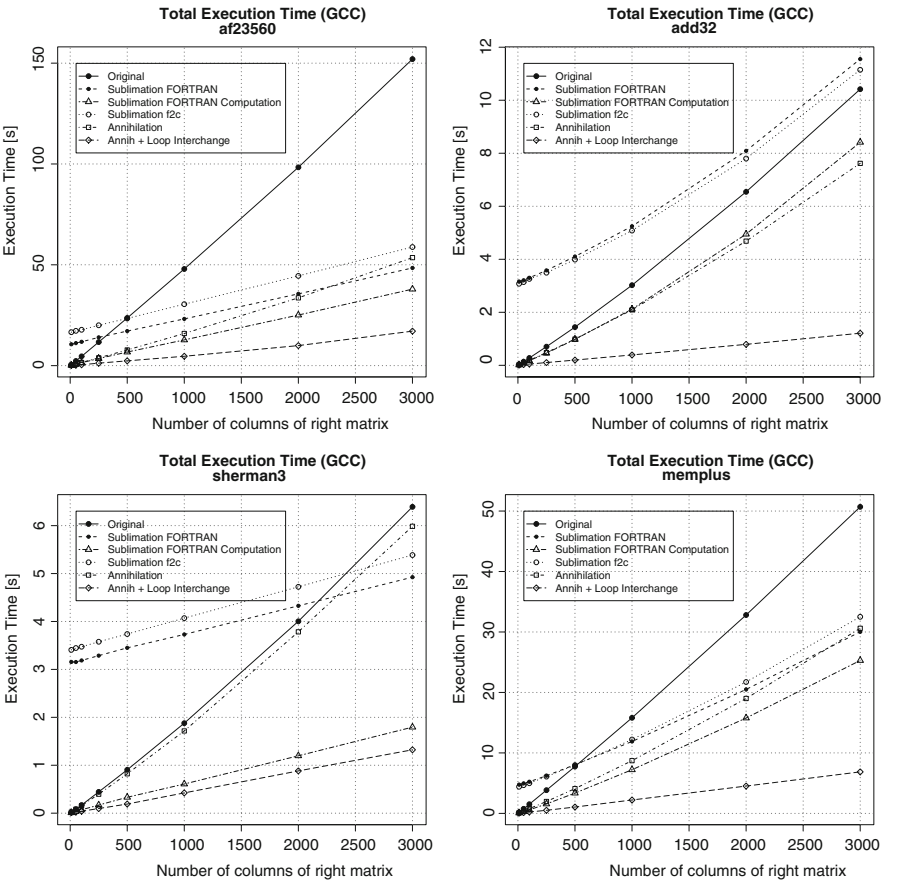

Fig. 7 Execution times using GNU GCC

The execution time of sublimation using FORTRAN is also plotted using the computation time only (initialization time left out) in order to compare the performance of the computation loop only. Note that if the initialization time is not taken into account, the code generated by sublimation shows a performance similar to that of annihilation. Especially the Intel FORTRAN compiler emits code that is close in performance to the code produced by annihilation. This shows that sublimation is a viable alternative if dependences prevent the application of annihilation.

Interestingly, for GCC it does not really matter whether the FORTRAN or the C version is used. For the Intel Compiler, in two of the four cases it makes quite a difference whether the FORTRAN or the C version is used. For the matrices *add32* and *memplus* the computation loop is considerably faster, if FORTRAN is used. In principle, the C code emitted by *f2c* should be the same, but apparently the FORTRAN code is easier to analyze by a compiler. The performance of the C versions is comparable for GCC and the Intel Compiler. Apparently, the Intel FORTRAN Compiler finds some optimization opportunities that the GCC FORTRAN compiler does not find. It can

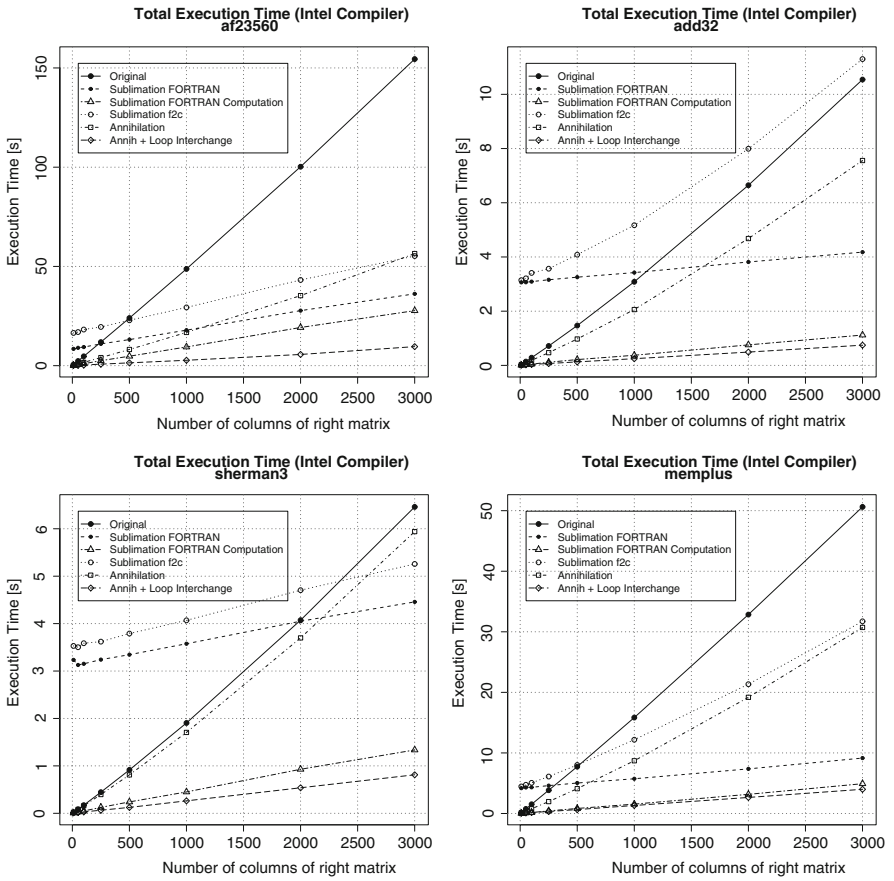

Fig. 8 Execution times using the Intel Compiler

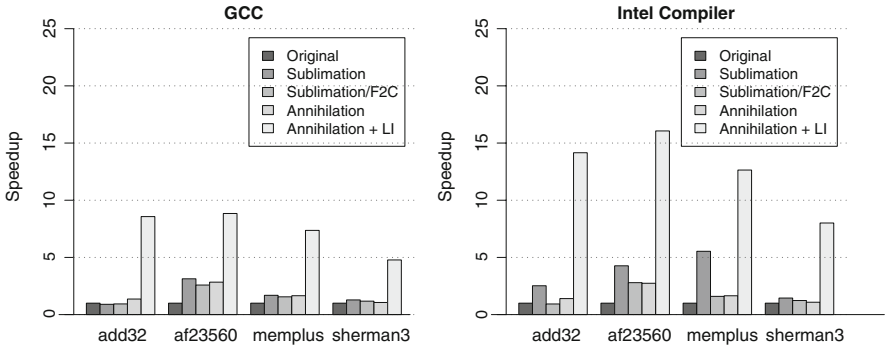

Fig. 9 Speedup obtained with 3000 result columns

Table 1 Mean initialization time in seconds

		af23560	add32	sherman3	memplus
Sublimation	GCC	10.439	3.130	3.124	4.688
	Intel	8.436	3.049	3.297	4.190
Sublimation/F2C	GCC	16.386	3.069	3.408	4.358
	Intel	16.344	3.146	3.482	4.419
Annihilation	GCC	0.073	0.006	0.005	0.029
	Intel	0.074	0.006	0.005	0.029

be concluded that sublimation does generate code which is optimizable, but that not every compiler fully exploits these opportunities.

5.2 Preconditioned Conjugate Gradient

There are cases where annihilation is not always the best option. Preconditioned conjugate gradient (as described by Golub and Van Loan [14]) is such an example. This algorithm iteratively solves linear systems. In each iteration, a matrix/vector multiplication is performed. The matrix is constant throughout execution, but the vector changes every iteration. Therefore, the code restructuring the vector cannot be hoisted outside of the containing loop. In addition, this vector is used in subsequent computations which of course expect the original access pattern. Therefore, the modified data must be copied in every iteration, to keep the restructured version up-to-date. In principle, this can be solved by restructuring all dependent data structures, but this has not been implemented yet. This does not mean that annihilation cannot be beneficial for this algorithm as restructuring might enable other optimizations, amortizing the cost of the restructuring.

First we will consider the application of sublimation. Sublimation has been applied to a C implementation of Conjugate Gradient with a diagonal preconditioner. The matrices used are *af23560*, *sherman3*, *memplus*, *impcol_b* and *lshp3466*. The algorithm does not converge, except for *sherman3*. This is not a problem, as we are interested in the program's behavior caused by the non-zero structure of the underlying problem, rather than the actual solution.

Table 2 shows the execution times of the two programs generated by MTC (our prototype compiler) performing sublimation. One version directly compiles the (at run-time) restructured FORTRAN code, the other uses *f2c* to convert the restructured code back to C (referred to as MTC/F2C). The execution times shown are the average execution times taken over 10 runs. The significance of the difference in execution times is determined using the *Student's t-test*. The threshold used for the *p-value* is 0.001. The only differences found to be significant are for the combination GCC with the matrix *impcol_b*. In the absolute sense, the differences are not very large and therefore the results for the version using *f2c* will not be considered further.

Figure 10 shows the execution times of PCG using the different matrices. The results for *lshp3466* are shown separately in Fig. 11, as for this matrix, the program shows some unexpected behavior. For the matrix *af23560*, the transformations performed

Table 2 Execution times for preconditioned conjugate gradient

Matrix	Compiler	Iterations	Total(pcg_mtc)	Total(pcg_mtc_f2c)	*p*-value
af23560	gcc	0	10.4369181	10.4565496	0.2332
		200	13.6794500	13.6856079	0.7277
		500	18.4899473	18.5281476	0.0928
		1000	26.5012467	26.5199758	0.5196
	icc	0	8.3799467	8.3820959	0.8543
		200	11.2647890	11.2720330	0.4640
		500	15.6034099	15.5978320	0.7378
		1000	22.8165724	22.8604394	0.3324
impcol_b	gcc	0	2.7630555	2.7659491	0.7064
		1000000	7.7020608	7.8059651	0.0000*
		2000000	12.6517297	12.8694533	0.0000*
		5000000	27.4455029	28.0216110	0.0000*
		10000000	52.1405536	53.2367686	0.0000*
	icc	0	2.7902089	2.7847962	0.7141
		1000000	5.7294116	5.7362139	0.1551
		2000000	8.6936637	8.6937868	0.9908
		5000000	17.6339414	17.5776914	0.5198
		10000000	32.6823472	32.3215858	0.1807
lshp3466	gcc	0	1.9389572	1.9393990	0.8050
		5000	3.1645367	3.1619932	0.9443
		10000	4.4255588	4.4025642	0.6569
		25000	8.2005100	8.1453305	0.7551
		50000	14.0215637	14.1054011	0.7226
	icc	0	1.9219801	1.9304343	0.0001
		5000	2.8958285	2.9275486	0.3830
		10000	3.9794020	3.8251235	0.1028
		25000	6.8309266	6.9324833	0.6294
		50000	11.4676299	11.6319217	0.4393
memplus	gcc	0	3.6738501	3.7176453	0.1371
		200	4.9385124	4.9246219	0.3406
		500	6.8018818	6.8135444	0.4001
		1000	9.9479365	9.8877520	0.0811
		2000	16.1671235	16.2084638	0.4743
	icc	0	3.1837252	3.1985074	0.6831
		200	4.1646083	4.2815048	0.1062
		500	5.5736581	5.5779907	0.7269
		1000	7.9800528	7.9539172	0.1920
		2000	12.7488172	12.7159817	0.5007
sherman3	gcc	0	3.1321917	3.1264834	0.7156
		200	3.2285829	3.2185971	0.1075
		500	3.3751111	3.3805490	0.6407
		1000	3.6319173	3.6358024	0.8249
		2000	4.1335331	4.1530528	0.7129
		2865	4.6117277	4.6734273	0.4756
	icc	0	3.0950571	3.0827892	0.4930
		200	3.1625665	3.1744411	0.2263
		500	3.3024061	3.3019816	0.9704
		1000	3.5116003	3.5093471	0.9205
		2000	3.9008675	3.8564076	0.2961
		2865	4.2025144	4.2786232	0.1312

MTC vs. MTC/F2C

* Indicates that the observed difference is significant

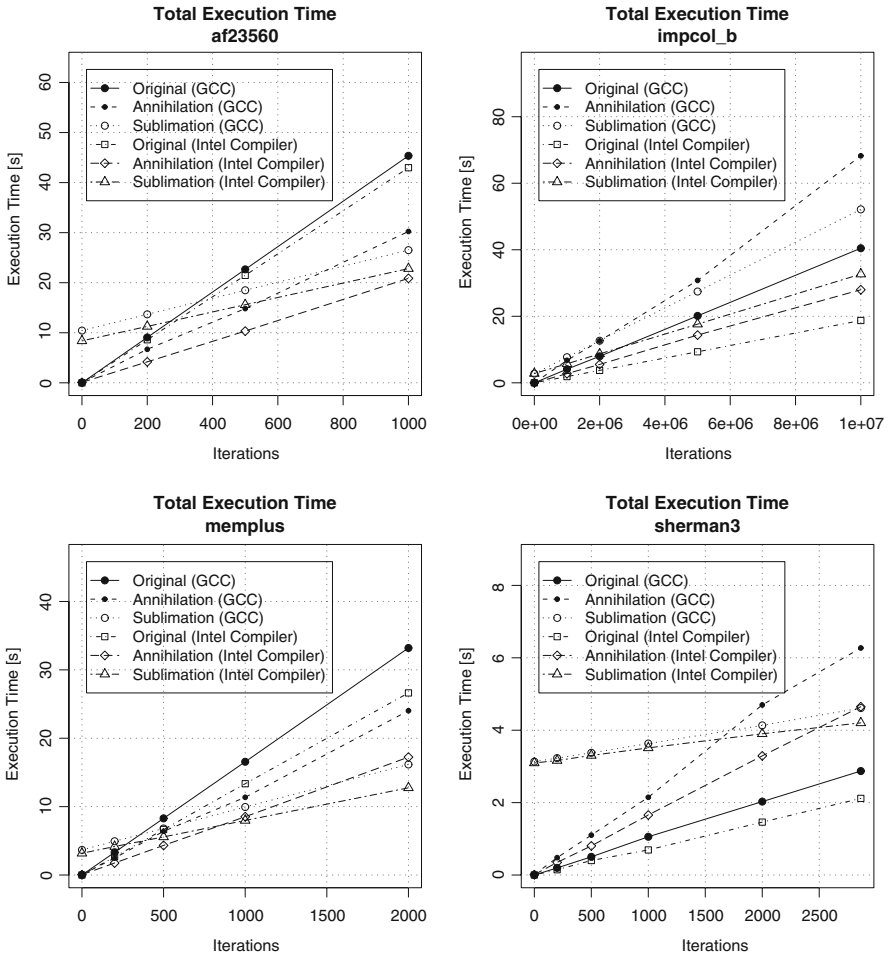

Fig. 10 Execution times for preconditioned conjugate gradient

by MTC are effective. Using both compilers, significant speedups are obtained. Both annihilation and sublimation result in faster code, sublimation being clearly superior in the asymptotic case. For *memplus*, the transformations are effective as well. Again, sublimation outperforms annihilation in the asymptotic case. PCG converges in 2875 iterations for *sherman3* and at this point the original code performs better than the versions produced by annihilation and sublimation. Note however that in the case of sublimation, the slope of the curve is less steep, which means that the computation itself is faster but it cannot compensate for the time spent in the initialization. The results are different for *impcol_b*. In this case, performance suffers if annihilation or sublimation is applied. This matrix is relatively small (53×53) and no appropriate access pattern can be found at run-time when sublimation is applied for this matrix, which results in the selection of a representation completely based on an indirectly addressed access storage scheme. This storage scheme performs worse than the linked

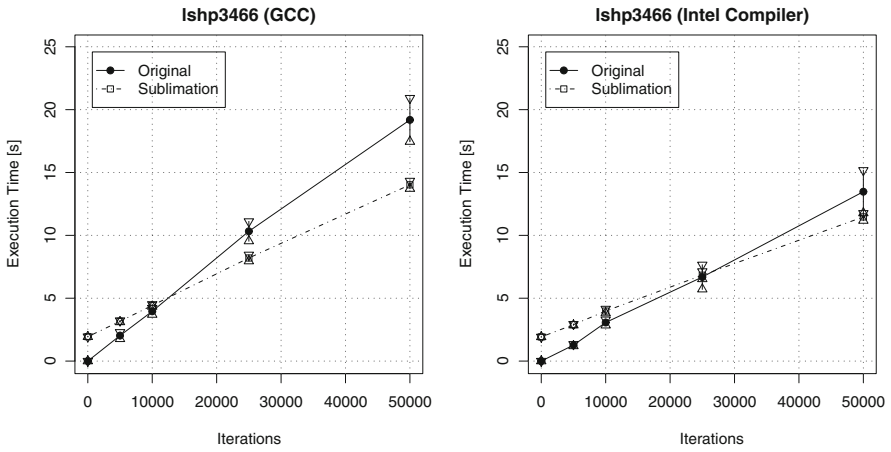

Fig. 11 Execution times for preconditioned conjugate gradient (lshp3466)

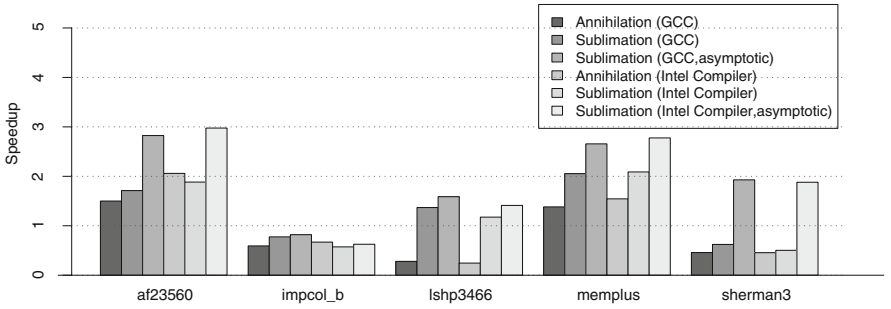

Fig. 12 Mean asymptotic speedup obtained by using MTC

list representation, which uses indirect addressing as well. The Intel Compiler clearly outperforms GCC on this matrix. For all other matrices in Fig. 10, the Intel Compiler performs slightly better than GCC.

As mentioned before, PCG shows some abnormal behavior when the matrix *lshp 3466* is used. Figure 11 shows the execution times for this matrix. In addition to the mean execution time, the 95% confidence intervals are also shown. Both for the Intel Compiler and GCC, there is a large variance in the execution times for the original version. Application of annihilation resulted in a performance degradation (Fig. 12).

Interestingly, the version using sublimation does not show this behavior and is likely to execute faster. This indicates that the transformations proposed in this paper do not only give gains in performance, but also can result in algorithms that show more robust behavior.

Table 3 compares the original algorithm with the version generated by MTC. In all cases (except for the matrix *lshp3466*) the differences in execution times are significant. Table 4 compares the performance of the Intel Compiler and GCC. From this data, it can be concluded that the Intel Compiler performs significantly better than GCC.

Table 3 Execution times for preconditioned conjugate gradient

Matrix	Compiler	Iterations	Total(pcg)	Total(pcg_mtc)	*p*-value
af23560	gcc	0	0.0014977	10.4369181	0.0000*
		200	9.0676140	13.6794500	0.0000*
		500	22.6633046	18.4899473	0.0000*
		1000	45.3536471	26.5012467	0.0000*
	icc	0	0.0011748	8.3799467	0.0000*
		200	8.6016974	11.2647890	0.0000*
		500	21.4728851	15.6034099	0.0000*
		1000	42.9741284	22.8165724	0.0000*
impcol_b	gcc	0	0.0000489	2.7630555	0.0000*
		1000000	4.1009672	7.7020608	0.0000*
		2000000	7.9636513	12.6517297	0.0000*
		5000000	20.1213715	27.4455029	0.0000*
		10000000	40.4568902	52.1405536	0.0000*
	icc	0	0.0000546	2.7902089	0.0000*
		1000000	1.8692082	5.7294116	0.0000*
		2000000	3.7324763	8.6936637	0.0000*
		5000000	9.3331125	17.6339414	0.0000*
		10000000	18.7334097	32.6823472	0.0000*
lshp3466	gcc	0	0.0002808	1.9389572	0.0000*
		5000	2.0282342	3.1645367	0.0000*
		10000	3.9577893	4.4255588	0.0058
		25000	10.3251169	8.2005100	0.0004
		50000	19.1854449	14.0215637	0.0002
	icc	0	0.0002627	1.9219801	0.0000*
		5000	1.2696215	2.8958285	0.0000*
		10000	3.0687554	3.9794020	0.0000*
		25000	6.6853483	6.8309266	0.7832
		50000	13.4729768	11.4676299	0.0470
memplus	gcc	0	0.0011539	3.6738501	0.0000*
		200	3.3156697	4.9385124	0.0000*
		500	8.2835206	6.8018818	0.0000*
		1000	16.5571594	9.9479365	0.0000*
		2000	33.1987006	16.1671235	0.0000*
	icc	0	0.0009095	3.1837252	0.0000*
		200	2.6729744	4.1646083	0.0000*
		500	6.6520123	5.5736581	0.0000*
		1000	13.3496854	7.9800528	0.0000*
		2000	26.6269574	12.7488172	0.0000*
sherman3	gcc	0	0.0003997	3.1321917	0.0000*
		200	0.1960706	3.2285829	0.0000*
		500	0.5022527	3.3751111	0.0000*
		1000	1.0568317	3.6319173	0.0000*
		2000	2.0277288	4.1335331	0.0000*
		2865	2.8716864	4.6117277	0.0000*
	icc	0	0.0003540	3.0950571	0.0000*
		200	0.1463178	3.1625665	0.0000*
		500	0.4011187	3.3024061	0.0000*
		1000	0.6927279	3.5116003	0.0000*
		2000	1.4612195	3.9008675	0.0000*
		2865	2.1161639	4.2025144	0.0000*

Original vs. MTC

*Indicates that the observed difference is significant

Table 4 Execution times for preconditioned conjugate gradient

Matrix	Program	Iterations	Total(ICC)	Total(GCC)	p-value
af23560	pcg	0	0.0011748	0.0014977	0.00000*
	pcg	200	8.6016974	9.0676140	0.00000*
	pcg	500	21.4728851	22.6633046	0.00000*
	pcg	1000	42.9741284	45.3536471	0.00000*
	pcg_mtc	0	8.3799467	10.4369181	0.00000*
	pcg_mtc	200	11.2647890	13.6794500	0.00000*
	pcg_mtc	500	15.6034099	18.4899473	0.00000*
	pcg_mtc	1000	22.8165724	26.5012467	0.00000*
impcol_b	pcg	0	0.0000546	0.0000489	0.00977
	pcg	1000000	1.8692082	4.1009672	0.00000*
	pcg	2000000	3.7324763	7.9636513	0.00000*
	pcg	5000000	9.3331125	20.1213715	0.00000*
	pcg	10000000	18.7334097	40.4568902	0.00000*
	pcg_mtc	0	2.7902089	2.7630555	0.08049
	pcg_mtc	1000000	5.7294116	7.7020608	0.00000*
	pcg_mtc	2000000	8.6936637	12.6517297	0.00000*
	pcg_mtc	5000000	17.6339414	27.4455029	0.00000*
	pcg_mtc	10000000	32.6823472	52.1405536	0.00000*
lshp3466	pcg	0	0.0002627	0.0002808	0.00001*
	pcg	5000	1.2696215	2.0282342	0.00023
	pcg	10000	3.0687554	3.9577893	0.00007*
	pcg	25000	6.6853483	10.3251169	0.00002*
	pcg	50000	13.4729768	19.1854449	0.00021
	pcg_mtc	0	1.9219801	1.9389572	0.00000*
	pcg_mtc	5000	2.8958285	3.1645367	0.00000*
	pcg_mtc	10000	3.9794020	4.4255588	0.00035
	pcg_mtc	25000	6.8309266	8.2005100	0.00000*
	pcg_mtc	50000	11.4676299	14.0215637	0.00000*
memplus	pcg	0	0.0009095	0.0011539	0.00000*
	pcg	200	2.6729744	3.3156697	0.00000*
	pcg	500	6.6520123	8.2835206	0.00000*
	pcg	1000	13.3496854	16.5571594	0.00000*
	pcg	2000	26.6269574	33.1987006	0.00000*
	pcg_mtc	0	3.1837252	3.6738501	0.00000*
	pcg_mtc	200	4.1646083	4.9385124	0.00000*
	pcg_mtc	500	5.5736581	6.8018818	0.00000*
	pcg_mtc	1000	7.9800528	9.9479365	0.00000*
	pcg_mtc	2000	12.7488172	16.1671235	0.00000*
sherman3	pcg	0	0.0003540	0.0003997	0.00000*
	pcg	200	0.1463178	0.1960706	0.00000*
	pcg	500	0.4011187	0.5022527	0.00000*
	pcg	1000	0.6927279	1.0568317	0.00000*
	pcg	2000	1.4612195	2.0277288	0.00000*
	pcg	2865	2.1161639	2.8716864	0.00000*
	pcg_mtc	0	3.0950571	3.1321917	0.11104
	pcg_mtc	200	3.1625665	3.2285829	0.00000*
	pcg_mtc	500	3.3024061	3.3751111	0.00000*
	pcg_mtc	1000	3.5116003	3.6319173	0.00006*
	pcg_mtc	2000	3.9008675	4.1335331	0.00050
	pcg_mtc	2865	4.2025144	4.6117277	0.00008*

Intel Compiler vs. GCC

* Indicates that the observed difference is significant

Speedups for the code produced by MTC have been obtained using the maximum number of iterations that data has been collected for. In addition, the asymptotic speedup for sublimation is calculated by leaving out the initialization times. All speedups are compared to the original code, using the same compiler. The results are shown in Fig. 12. The asymptotic speedup shows that the code produced by MTC is potentially faster than the original code in all cases, except for *impcol_b*. It also outperforms code resulting from annihilation in nearly all cases (except for *impcol_b* when using the Intel Compiler).

6 Future Work and Conclusions

The results obtained from the experiments suggest that the transformations proposed in this paper can be very effective. Code containing pointer chains are hard to analyze and transforming such codes to regularized codes can have major benefits. The methods proposed here result in a code that has the following properties. (1) Loop interchange is enabled and related data exhibits spatial locality due to linearization and structure splitting. (2) Structure splitting reduces the amount of data that must be fetched from memory because data that is never needed can be loaded in the CPU cache if the linked list itself was used in the computation. (3) Loop extraction results in simpler computation loop structures which are easier to analyze. (4) After loop extraction the expensive traversal of a linked list is less deeply nested, removing many unnecessary traversals of the linked list. Together, these properties enable major performance improvements.

The choice between annihilation and sublimation involves a trade-off. Annihilation is relatively simple and does not need run-time support, sublimation is more complicated and has significant run-time overhead. Table 1 showed how large this difference can be. However, as the results from Sect. 5 show, sublimation can be very efficient, depending on both the dependences which are present in the code and the underlying data structure. Dependences may prevent moving the restructuring code across loops, favoring another access pattern to be used for restructuring. Such trade-offs are a topic for future research.

Although it is well known that the choice of data structures has a big influence on the performance of algorithms, most compiler transformations do not directly target data structures but only consider the computational code. One of the reasons for this is that especially pointer structures impose severe restrictions on standard transformations. In this paper various transformations are described which directly target the transformation of data structures and it is shown that substantial speedups can be achieved.

We fully realize that the techniques as described in this paper are very ambitious and still require a large research and development effort to be fully implemented in a production type compiler. Also the code samples used in this paper only reflect algorithm/kernel based codes and the question remains how the techniques will expand if full application codes were to be considered. In principle, non-numerical algorithms can be handled as well, but the gains in computation speed must compensate for the initialization overhead.

For full applications, there are various problems which need to be dealt with. These issues will be addressed in future publications. Here, we give a short overview of some topics that need further elaboration.

Currently, only relatively simple pointer traversals (linked list) are supported. In future work, we plan to extend our techniques such that more complicated traversals can be recognized and transformed to a dense representation. A good starting point for more complicated traversal patterns is described by Hummel et al. [17], who define possible access paths encountered during execution. This is a flexible approach that also enables our techniques on cyclic data structures, as long as the traversal pattern at run-time remains acyclic.

Modifications of data structures that have an impact on the traversal pattern are not taken into account. This prevents the application of our techniques in many cases, which potentially could benefit from them. Therefore, more research must be done on how to permit changes of the traversal patterns. Additionally, the methods should be expanded to include interprocedural application of our techniques, such that restructuring overhead can be moved across function boundaries. A program wide approach also allows for reuse of restructured and recompiled codes, further reducing overhead.

The restructuring techniques annihilation and sublimation must be taken to a higher level. Multiple different access patterns (possibly indirect) should be supported and modification of these elements must be made possible. This requires a write-back phase in which the modified data members are written back in their original location. The access patterns should also be propagated throughout the entire application, such that other data structures can also be remapped, possibly eliminating the need for the write-back stage.

Actually, the work as described in this paper might be seen as a first step in realizing data structure independent programming. Weng et al. [31] made an attempt to separate problem specification and and the actual data structures used for multi-dimensional scientific datasets. Langella et al. use a distributed storage middleware to obtain location independence [21]. All these approaches focus on a high level of granularity. Wijshoff [32] calls for generic approaches to eliminate data structure dependence, such that it is the compiler that is responsible for choosing good implementations. All these views have in common that they aim to hide the actual data layout from the core application.

The approach taken in this paper hopefully will lead to a new direction of devising future implementations of compilers as well as applications. In this case, future applications could be developed regardless of specific data structure choices. Thereupon, advanced compilers will transform these applications into applications which are optimized for specific data structure instances. This will not only improve efficiency considerably but will also provide a simple and transparent means of defining applications.

References

1. Bender, M.A., Hu, H.: An adaptive packed-memory array. In: PODS '06: Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 20–29. ACM Press, New York, NY, USA (2006)
2. Bik, A.J.C. (ed.): Compiler Support for Sparse Matrices, PhD Thesis. Leiden University, The Netherlands (1996)
3. Bik, A.J.C., Knijenburg, P.M.W., Wijshoff, H.A.G.: Reshaping access patterns for generating sparse codes. In: LCPC '94: Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing, pp. 406–420. Springer-Verlag, London, UK (1994)
4. Bik, A.J.C., Wijshoff, H.A.G.: Advanced compiler optimizations for sparse computations. J. Parallel Distrib. Comput. **31**(1), 14–24 (1995)
5. Bik, A.J.C., Wijshoff, H.A.G.: Automatic data structure selection and transformation for sparse matrix computations. IEEE Trans. Parallel Distrib. Syst. **7**(2), 109–126 (1996)
6. Chilimbi, T.M., Davidson, B., Larus, J.R.: Cache-conscious structure definition. In: PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, pp. 13–24. ACM Press, New York, NY, USA (1999)
7. Chong, S., Rugina, R.: Static analysis of accessed regions in recursive data structures. In: Static Analysis, 10th International Symposium, SAS 2003, pp. 463–482 (2003)
8. Duff, I.S., Grimes, R.G., Lewis, J.G.: Users' guide for the Harwell-Boeing sparse matrix collection (Release I). Technical Report RAL 92-086, Chilton, Oxon, England (1992)
9. Feldman, S.I., Gay, D.M., Maimone, M.W., Schryer, N.L.: A FORTRAN to C converter. SIGPLAN Fortran Forum **9**(2), 21–22 (1990)
10. Ghiya, R.: Putting Pointer Analysis to Work, PhD Thesis. School of Computer Science, McGill University, Montreal, Canada (1998)
11. Ghiya, R., Hendren, L.J.: Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In: POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 1–15. ACM, New York, NY, USA (1996)
12. Ghiya, R., Hendren, L.J., Zhu, Y.: Detecting parallelism in C programs with recursive data structures. In: CC '98: Proceedings of the 7th International Conference on Compiler Construction, pp. 159–173. Springer-Verlag, London, UK (1998)
13. Golovanevsky, O., Zaks, A.: Struct-reorg: current status and future perspectives. In: Proceedings of the GCC Developers' Summit, pp. 47–56. http://www.gccsummit.org (2007)
14. Golub, G.H., Van Loan, C.F.: Matrix Computations. Johns Hopkins University Press, Baltimore (1989)
15. Groot, S., van der Spek, H.L.A., Bakker, E.M., Wijshoff, H.A.G.: The automatic transformation of linked list data structures (abstract). In: PACT '07: Proceedings of the 16th Intenational Conference on Parallel Architectures and Compilation Techniques. IEEE Computer Society, Washington, DC, USA (2007)
16. Hagog, M., Tice, C.: Cache aware data layout reorganization optimization in GCC. In: Proceedings of the GCC Developers' Summit, pp. 69–92. http://www.gccsummit.org (2005)
17. Hummel, J., Hendren, L.J., Nicolau, A.: A general data dependence test for dynamic, pointer-based data structures. In: PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language Design and Implementation, pp. 218–229. ACM Press, New York, NY, USA (1994)
18. Hwang, Y.-S., Saltz, J.: Interprocedural definition-use chains of dynamic pointer-linked data structures. Sci. Program. **11**(1), 3–37 (2003)
19. Karlsson, M., Dahlgren, F., Stenström, P.: A prefetching technique for irregular accesses to linked data structures. In: Proceedings of the Sixth International Symposium on High-Performance Computer Architecture, 8–12 January 2000, Toulouse, France, pp. 206–217. IEEE Computer Society, Washington, DC, USA (2000)
20. Kodukula, I., Pingali, K.: Data-centric transformations for locality enhancement. Int. J. Parallel Prog. **29**(3), 319–364 (2001)
21. Langella, S., Hastings, S., Oster, S., Kurc, T., Catalyurek, U., Saltz, J.: A distributed data management middleware for data-driven application systems. In: CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing, Washington, DC, USA, pp. 267–276. IEEE Computer Society (2004)

22. Lattner, C., Adve, V.: Automatic pool allocation for disjoint data structures. In: MSP '02: Proceedings of the 2002 Workshop on Memory System Performance, pp. 13–24. ACM, New York, NY, USA (2002)
23. Lin, Y., Padua, D.: Compiler analysis of irregular memory accesses. In: PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, pp. 157–168. ACM Press, New York, NY, USA (2000)
24. Luk, C.-K., Mowry, T.C.: Compiler-based prefetching for recursive data structures. In: ASPLOS-VII: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 222–233. ACM Press, New York, NY, USA (1996)
25. NIST. Matrix Market. http://math.nist.gov/MatrixMarket (2007)
26. Padua, D.A., Wolfe, M.J.: Advanced compiler optimizations for supercomputers. Commun. ACM **29**(12), 1184–1201 (1986)
27. Rubin, S., Bernstein, D., Rodeh, M.: 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, pp. 259–273. Springer-Verlag, London, UK (1999)
28. Rugina, R., Rinard, M.C.: Symbolic bounds analysis of pointers, array indices, and accessed memory regions. ACM Trans. Program. Lang. Syst. **27**(2), 185–235 (2005)
29. Silvius, R., Rauchwerger, L., Hoeflinger, J.: Hybrid analysis: static & dynamic memory reference analysis. Int. J. Parallel Prog. **31**(4), 251–283 (2003)
30. Saltz, J.H., Mirchandaney, R., Crowley, K.: Run-time parallelization and scheduling of loops. IEEE Trans. Comput. **40**(5), 603–612 (1991)
31. Weng, L., Agrawal, G., Catalyurek, U., Kurc, T., Narayanan, S., Saltz, J.: An approach for automatic data virtualization. In: HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC '04) pp. 24–33. IEEE Computer Society, Washington, DC, USA (2004)
32. Wijshoff, H.A.G.: Programming without bothering about data structures? IEEE Comput. Sci. Eng. IEEE Trans. Comput. (3), 67–68 (1996)
33. Yang, C.-L., Lebeck, A.R.: Push vs. pull: data movement for linked data structures. In: ICS '00: Proceedings of the 14th International Conference on Supercomputing, pp. 176–186. ACM Press, New York, NY, USA (2000)
34. Zhao, L., Wijshoff, H.A.G.: A case study in automatic data structure selection for optimizing sparse matrix computations. In: Proceedings of the IEEE International Workshop on Advanced Compiler Technology for High Performance and Embedded Systems (IWACT), July 2001, Bucharest, Romania, pp. 22–55. Editura MATRIX ROM (2001)
35. Zhong, Y., Orlovich, M., Shen, X., Ding, C.: Array regrouping and structure splitting using whole-program reference affinity. In: PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, pp. 255–266. ACM Press, New York, NY, USA (2004)