Forelem: A Versatile Optimization Framework For Tuple-Based Computations

K. F. D. Rietveld and H. A. G. Wijshoff

Leiden University, LIACS, Leiden, The Netherlands
krietvel@liacs.nl, harryw@liacs.nl

Abstract. The *forelem* framework arose from the unification of code optimization in seemingly distinct fields of programming: transactional (database) applications and other (imperative) applications. Traditionally, the optimization of transactional applications for a large part relies on DBMS (query) optimizations to efficiently retrieve the desired data from a database. On the other hand, traditional optimizations on application codes rely for a large part on (loop based) compiler optimizations, which are vital for the generation of efficient machine code. In this paper, we will make a case for the *forelem* framework to be used as a versatile unifying optimization platform for tuple-based computations in general. An overview is presented of all transformations that are included within the *forelem* framework up till now. Apart from transformations that target the loop structure, the *forelem* framework is also capable of deriving efficient data storage formats for a given computation.

Keywords: Optimizing Compilers, Intermediate Representation, Tuple-Based Computations, Program Transformation

1 Introduction

The *forelem* framework arose from the unification of code optimization in seemingly distinct fields of programming: transactional (database) applications and other (imperative) applications, and unifies these distinct fields of programming by expressing queries in an intermediate representation as a series of tuple accesses governed by simple loop control. Subsequently, this intermediate representation is optimized by traditional optimizing compiler techniques, accomplishing results similar to query optimization.

In past work, we have already shown the advantages of the approach taken by the *forelem* framework. The *forelem* framework has been used to perform vertical integration of database applications, where queries in a database application are replaced with code segments that evaluate these queries using direct access to a local data store [13,14]. Subsequently, the application and data access codes are optimized together and we have shown that this can result in a reduction in energy consumption up to 90% [13]. Furthermore, possibilities to optimize Big Data applications using parallel *forelem* loops have been investigated as well as

a novel approach for the optimization of Sparse Matrix kernels, that leads to the automatic generation of different storage formats for the sparse structures.

The main feature of the *forelem* framework consists of accessing data through a tuple space. As the *forelem* framework was initially envisioned for database applications, its main features rely on viewing data as being stored as (multi)sets of tuples. Next to accessing data as tuples, the *forelem* framework allows the execution order of tuple computations (transactions) to be out of order. Because of this out of order execution, application of compiler optimizations has to be carefully handled. As standard compiler optimizations rely on data dependence analysis and loop-carried dependencies, and these loop-carried dependencies are non-existing in *forelem* loop nests, the conditions under which the transformations can be applied have to be reconsidered.

Within the *forelem* framework, compiler transformations are being defined which operate on three levels: the tuple level, the materialized loop index level and the concretized data access level. The *forelem* tuple level provides an elegant representation method for expressing different data access codes such as database queries and sparse matrix algebra. Within the materialized loop index level, index sets on the tuple space that specify access patterns, are being represented as array accesses. By giving the compiler transformations framework access to this second level of data access, the compiler can address the order of data access while the order of execution is not specified. Finally, within the concretized data access level, loops are expressed using regular (integer) iteration bounds. At this level, standard compiler optimizations can be applied, taking into account the different semantics for data dependencies.

For the different application areas of the *forelem* framework, different transformations have been devised. While such transformations were defined within the context of a particular application area, these transformations are generic in nature and may well be of use in other application areas. In this paper, an overview is presented of all transformations that are included within the *forelem* framework up till now. Conditions under which the transformations can be applied, with regards to the different treatment of data dependencies, will be discussed.

This paper is organized as follows: in Section 2 the *forelem* intermediate representation is introduced. Section 3 describes basic code transformations that can be applied at the tuple level and the conditions under which these transformations can be applied. Section 4 discusses loop scheduling and data decomposition techniques that are used for the parallelization of *forelem* loops. Section 5 introduces the orthogonalization transformation, that can be used to impose a certain order on the iteration of the data. This is a preparatory step to materialization, discussed in Section 6. In this section, the process of transforming a loop to the materialized loop index level is defined and several transformations applicable at the materialized loop index level are described. Section 7 outlines how loops are converted to the concretized data access level. Section 8 briefly presents how the *forelem* framework is implemented. Finally, Section 9 presents the conclusions and future work.

2 The Forelem Intermediate Representation

In this section the basics of the *forelem* intermediate representation are described. The intermediate representation is centered around the *forelem* loop construct. Each *forelem* loop iterates over a specific array of structures. The subscripts of this array that are accessed are fetched from an "index set" that is associated with the array.

The arrays of structures that are iterated by *forelem* loops are modeled after database tables which are defined as multisets. The structure reflects the format of a database tuple. In an array of structures `A` a tuple at index `i` is accessed with `A[i]` and a specific field *field1* in that tuple is accessed with `A[i].field1`.

An *index set* is a set containing subscripts $\mathtt{i} \in \mathbb{N}$ into an array. Since each array subscript is typically processed once per iteration of the array, these subscripts are stored in a regular set. Index sets are named after the array they refer to, prefixed with "p". For example, `pA` is the index set of all subscripts into an array `A`: $\forall s \in \mathtt{A} : \exists \mathtt{i} \in \mathtt{pA} : \mathtt{A}[\mathtt{i}] = s$. Random access of an index set by subscript is not possible, instead all accesses are done using the $\in$ operator.

The body of a *forelem* loop typically performs an action on the tuple subscripted by the current value of the loop iterator. When used in the context of database codes, the loop body often outputs tuples to a temporary or result set. Temporary sets are generally named $\mathscr{T}_1, \mathscr{T}_2, ..., \mathscr{T}_n$ and result sets $\mathscr{R}_1, \mathscr{R}_2, ..., \mathscr{R}_n$. In the context of, for example, sparse matrix codes a computation is typically performed also involving data from dense matrices or vectors. Results could be stored in a dense array.

Considering an array `A` with fields *field1* and *field2*, a *forelem* loop that iterates all entries of `A`, outputting the value of *field1* of each row, is written as follows:

```
forelem (i; i ∈ pA)
  ℛ = ℛ ∪ (A[i].field1)
```

Although the *forelem* loop appears to be very similar to a *foreach* loop that exists in many common programming languages, *forelem* loops distinguish themselves by the use of the index sets. Every *forelem* loop iterates a single array, using subscripts from an index set that is associated with that array. Note that, the order of the subscripts in the index set is undefined. As such, *forelem* loops do not have explicit looping structures and the exact semantics of the iteration of an array are determined in the course of the optimization process. Index sets are the essence of *forelem* loop nests as they encapsulate iterations and simplify the loop control so that aggressive compiler optimizations can be successfully applied.

Using conditions on index sets it is possible to narrow down the range of the array that is iterated. For example, the index set denoted by `pA.field2[k]` contains only those subscripts into `A` for which *field2* has value `k`. This is expressed mathematically as follows:

$$\mathtt{pA.field2[k]} \equiv \{\mathtt{i} \mid \mathtt{i} \in \mathtt{pA} \wedge \mathtt{A[i].field2} = \mathtt{k}\}$$

So, in order to only iterate entries of `A` in which the value of *field2* is 10, the following *forelem* loop is used:

```
forelem (i; i ∈ pA.field2[10])
  ℛ = ℛ ∪ (A[i].field1)
```

Note, that `pA.field2[10]` is not expressed more explicitly as the exact execution will be determined by the optimization process. This index set might be explicitly generated (at compile- or run-time), combined with other index sets, moved or eliminated. Alternatively, during the optimization process it may be decided to create a variant of array `A` only containing the tuples to be iterated.

More sophisticated index sets are possible, such as having conditions on multiple fields, in this case on *field1* and *field2*:

$$\texttt{pA.(field1, field2)}[(\texttt{k}_1, \texttt{k}_2)] \equiv \{\texttt{i} \mid \texttt{i} \in \texttt{pA} \wedge \texttt{A[i].field1} = \texttt{k}_1 \wedge \texttt{A[i].field2} = \texttt{k}_2\}$$

Instead of a constant value, the values $\texttt{k}_n$ can also be a reference to a value from another array. To use such a reference, the array, subscript into the array and field name must be specified, e.g.: `A[i].field`. To select values `field1` > 10 an interval is used: $(10, \infty)$.

3 Basic Code Transformations

In this section, the different basic code transformations that can be applied on the *forelem* tuple level are described. These transformations are based on existing optimizing compiler techniques. Therefore, the description of the transformations will refer to common compiler analysis techniques: data dependence analysis [10,12,3,18] and def-use analysis [2,8].

3.1 Loop Invariant Code Motion

Loop Invariant Code Motion is a kind of common subexpression elimination where statements which are invariant under the loop's iteration variable can be moved to an outer loop or completely out of the loop nest. Within the *forelem* framework this transformation is generally used to move condition testing of array fields to outer loops to prune the iteration space, or to inner loops to not inhibit or complicate the application of loop transformations. For example:

```
forelem (i; i ∈ pX)
  forelem (j; j ∈ pY)
    if (X[i].field2 == value && Y[j].field2 == X[i].field1)
      ℛ = ℛ ∪ (Y[j].field1)
```

compares the value `X[i].field2` with a constant `value`. The reference `X[i].field2` is invariant under the inner loop, so can be moved to the outer loop. Fully moving the condition test out of the loop nest is not possible, because the array reference is variant under the outermost loop. The result is:

```
forelem (i; i ∈ pX)
  if (X[i].field2 == value)
    forelem (j; j ∈ pY)
      if (Y[j].field2 == X[i].field1)
        ℛ = ℛ ∪ (Y[j].field1)
```

Similarly, statements can be moved to the innermost loop, to enable more possibilities for the application of loop transformations, such as Loop Interchange.

3.2 Loop Interchange

The standard Loop Interchange transformation changes the order in which the statements in the loop are executed. This transformation is only valid if the new execution order preserves all dependencies of the original execution order [18]. Commonly, data-dependence analysis [10,12,3] is employed to formally verify whether the data-dependence relations are preserved across loop transformations. In general, only certain loop-carried dependencies can prevent application of Loop Interchange. A *forelem* loop does not specify a particular execution order and therefore loop-carried dependencies cannot exist. As a consequence, interchanges of loops in a perfect loop nest are always valid.

Loop-carried dependencies are therefore only caused by dependencies of the loop bounds of inner loops on outer loop iteration counters. In this case, Loop Invariant Code Motion is first used to move the conditions to the inner loop before the loop nest is reordered and back to the outermost loop after the reordering. This way, Loop Interchange is applied to a perfectly nested loop nest.

Within the *forelem* framework the Loop Interchange transformation is used to reorder loops such that as many conditions as possible are tested in the outermost loop to prune the search space. As an example, consider:

```
forelem (j; j ∈ pY)
  forelem (i; i ∈ pX.(field1,field2)[(Y[j].field2,value)])
    ℛ = ℛ ∪ (Y[j].field1)
```

First, the conditions are made explicit and, if necessary, moved to the outermost loop. Now that the loop nest is in a perfectly nested form, this allows the two loops to be interchanged:

```
forelem (i; i ∈ pX)
  forelem (j; j ∈ pY)
    if (X[i].field2 == value && Y[j].field2 == X[i].field1)
      ℛ = ℛ ∪ (Y[j].field1)
```

3.3 Iteration Space Expansion

Within the *forelem* framework a transformation known as Iteration Space Expansion is defined. This transformation is inspired by the Scalar Expansion transformation, which is typically used to enable parallelization of loop nests, and by the expansion of the iteration spaces iterated by loops, as described in [17], used to transform irregular access patterns into regular ones.

Iteration Space Expansion expands the iteration space of a *forelem* loop by removing conditions on its index set. For a loop of the form, with `SEQ` denoting a sequence of statements:

```
forelem (i; i ∈ pA.field[X])
  SEQ;
```

the following steps are performed:

1. the condition `A[i].field == X` is removed, which expands the iteration space so that the entire array `A` is visited,
2. scalar expansion is applied on all variables that are written to in the loop body denoted by *SEQ* and references to these variables are subscripted with the value tested in the condition, in this case `A[i].field`,
3. all references to the scalar expanded variables after the loop are rewritten to reference subscript `X` of the scalar expanded variable.

3.4 Loop Fusion

Loop Fusion [9] is a traditional compiler optimization that can be readily applied to *forelem* loops. The transformation can, under certain conditions, merge two loops (at the same level if contained in a larger loop nest) into a single loop. Application of Loop Fusion is only prohibited by certain loop-carried dependencies. Such loop-carried dependencies do not exist in *forelem* loops. Therefore, Loop Fusion can be applied on two adjacent *forelem* loops if the iteration spaces of the two loops are equal. This is the case if the index sets for both loops refer to the same table and contain the same set of subscripts into these tables. After Loop Fusion has been applied, the bodies of both loops are then executed for the same set of subscripts into the same array. For example:

```
forelem (i; i ∈ pTable1)
  ℛ1 = ℛ1 ∪ (Table1[i].field1)
forelem (i; i ∈ pTable1)
  ℛ2 = ℛ2 ∪ (Table1[i].field2)
```

can be rewritten into the following, because of the equal iteration bounds:

```
forelem (i; i ∈ pTable1)
{
  ℛ1 = ℛ1 ∪ (Table1[i].field1)
  ℛ2 = ℛ2 ∪ (Table1[i].field2)
}
```

Note, that *forelem* loops generally only access the array being iterated using the subscript of the current iteration. E.g., an access into an array always has the form i and not $i + 2$ or similar. As a consequence, a condition preventing Loop Fusion from being applied will in general not occur.

4 Parallel Forelem Loops

Within the *forelem* framework, parallelization consists out of loop scheduling, which is the problem of scheduling a parallel loop's iterations onto the available processors, and data distribution (or decomposition) to the processors. Loop scheduling is implemented through the application of Loop Blocking to the iteration space of a *forelem* loop. A distinction is made between *direct* and *indirect* loop scheduling, both of which will be described in this section. Finally, it is shown how the data set can be decomposed based on the created loop schedule.

Loop scheduling follows from the application of Loop Blocking to the iteration space of a *forelem* loop. With *direct* loop scheduling, the iteration space is blocked by partitioning the index set that is iterated by the *forelem* loop. On the other hand, *indirect* loop scheduling is achieved by blocking on the value range of a field in the accessed array.

As an example, consider an array `A` with fields *field1* and *field2*, and the following loop where *SEQ* denotes a sequence of statements:

```
forelem (i; i ∈ pA)
  SEQ;
```

In order to parallelize this loop to N processors, a loop schedule must be created. To create a direct loop schedule, Loop Blocking splits the iteration space of this loop, which is the index set `pA`, into N partitions:

$$\mathtt{pA} = \mathtt{p}_1\mathtt{A} \cup \mathtt{p}_2\mathtt{A} \cup \ldots \cup \mathtt{p}_N\mathtt{A}$$

and the *forelem* loop becomes:

```
for (k = 1; k <= N; k++)
  forelem (i; i ∈ p_kA)
    SEQ;
```

Subsequently, to parallelize this loop to N processors, each processor must be assigned a partition of the index set `pA`. This is achieved by replacing the *for* loop with a *forall* loop, indicating that the outer loop is executed in parallel:

```
forall (k = 1; k <= N; k++)
  forelem (i; i ∈ p_kA)
    SEQ;
```

As a next step, the data can be decomposed according to the selected partitioning. So, a decomposition of table `A` is created:

$$\mathtt{A} = \mathtt{A}_1 \cup \mathtt{A}_2 \cup \ldots \cup \mathtt{A}_N$$

based on the partitioned index sets $\mathtt{p}_k\mathtt{A}$. Note that, this decomposition of **A** yields an index set $\mathtt{pA}_k$ for every $\mathtt{A}_k$. The loop operating on the decomposed data is:

```
forall (k = 1; k <= N; k++)
  forelem (i; i ∈ pA_k)
    SEQ;
```

where in the loop body data is accessed through for example $\mathtt{A}_k[\mathtt{i}].\mathtt{field1}$. Note that, in case data accesses are performed to data that is not available locally after the data decomposition, these accesses can be resolved by performing remote communication to a processor that does have the necessary data available.

In indirect data partitioning, Loop Blocking is not done based on the iterated index set, but on the value range of one of the table's accessed fields. Consider the same starting point:

```
forelem (i; i ∈ pA)
  SEQ;
```

Array **A** is to be distributed into N partitions based on *field1*. The notation *A.field1* denotes the set of values of the *field1* found in all subscripts of **A**. If `X = A.field1`, then

$$\mathtt{X} = \mathtt{X}_1 \cup \mathtt{X}_2 \cup \ldots \cup \mathtt{X}_N$$

is a partitioning of **X** into N segments. The blocked loop is:

```
for (k = 1; k <= N; k++)
  for (l ∈ X_k)
    forelem (i; i ∈ pA.field1[l])
      SEQ;
```

In this loop nest the outer loop can be parallelized. In the parallelized loop nest a processor P_k is responsible for processing partition $\mathtt{X}_k$ of this partitioning and will execute the original *forelem* loop only for $i \in \mathtt{pA}, l \in \mathtt{X}_k : \mathtt{A}[\mathtt{i}].\mathtt{field1} = l$. This results in:

```
forall (k = 1; k <= N; k++)
  for (l ∈ X_k)
    forelem (i; i ∈ pA.field1[l])
      SEQ;
```

Also in this case, the table **A** can be decomposed based on the selected *indirect* loop schedule. The decomposition of **A** into N parts $\mathtt{A}_k$, with corresponding index sets $\mathtt{pA}_k$ is based on the partitioning **X** into $\mathtt{X}_k$. This results in the following loop nest:

```
forall (k = 1; k <= N; k++)
  for (l ∈ X_k)
    forelem (i; i ∈ pA_k.field1[l])
      SEQ;
```

where the loop body accesses, for example, $\texttt{A}_k[\texttt{i}].\texttt{field1}$. Note that, this data decomposition guarantees that $\texttt{pA}_k$ only contains subscripts `i` such that values $\texttt{A}_k[\texttt{i}].\texttt{field1}$ are always contained in $\texttt{X}_k$. Based on this observation, the loop can be simplified to:

```
forall (k = 1; k <= N; k++)
  forelem (i; i ∈ pA_k)
    SEQ;
```

without affecting the final result.

Within the *forelem* framework, the optimization of the data distribution is performed after the selection and optimization of the data partitioning or loop scheduling. The process of data distribution optimization depends on the communication model that is used to transfer data between processors, on any initial data distribution that is present and on the loop schedules that have been selected for other *forelem* loops in the application that access the same data. Details of how data distribution and the communication model are represented in the intermediate representation and how this optimization process is carried out will be described in a future work.

Many static and dynamic approaches to loop scheduling have been described in the literature [11,16,5]. A static loop schedule is determined entirely at compile-time. Dynamic approaches schedule iterations to idle processors at run-time and have the opportunity to better balance the load in case the cost for each loop iteration is not equal.

An example dynamic scheduling approach is Guided Self-Scheduling (GSS) [11]. In GSS, iterations of loops are scheduled to idle processors at runtime. Iterations are allocated in groups called chunks. The process starts with a large chunk size and this size gradually decreases with the course of execution. The next chunk size to use is determined by dividing the number of remaining iterations by the number of processors. Processors that finish their chunk earlier than other processors are assigned a new smaller chunk. This technique results in a better balancing of the work.

5 Orthogonalization

In *forelem* loops, iteration of a table of tuples is controlled by the index set. No order is defined on the index set, which has as consequence that the iteration order of the table is undefined. In this section, the *orthogonalization* transformation is introduced, which makes it possible to impose a certain order in which the table is iterated. This is achieved by partitioning the accesses to the array based on the values of one or more table fields. The orthogonalization transformation

is used to control the order in which data is accessed as a preparatory step to Materialization, which is discussed in the next section.

Let `A` be a table with `field1`, `field2`, ...`fieldn`. Consider the loop:

```
forelem (i; i ∈ pA)
  ... A[i] ...
```

In this loop, the tuples of `A` can be iterated in any order. As an example, assume an iteration order is to be imposed on `A` such that tuples `A` are accessed in blocks with equal values for `field1`. The orthogonalization transformation is carried out to achieve this, resulting in the following loop nest:

```
forelem (ii; ii ∈ A.field1)
  forelem (i; i ∈ pA.field1[ii])
    ... A[i] ...
```

`A.field1` in the outer loop denotes all possible values of `field1` that occur in `A`. So, the iteration space of the outer loop consists out of every value of `field1` in `A`.

The original loop iterates all tuples of `A`. The transformed loop nest will for every value of `field1`, iterate all tuples of `A` for which `field1` equals this value. As a result, the transformed loop also iterates all tuples of `A`.

Application of the orthogonalization transformation is not limited to a single field. An example of orthogonalization on two fields is:

```
forelem (ii; ii ∈ A.field1)
  forelem (jj; jj ∈ A.field2)
    forelem (i; i ∈ pA.(field1,field2)[(ii,jj)])
      ... A[i] ...
```

The outer loops that are introduced by the orthogonalization transformation iterate all values of a given table field. If it is possible to express this range of values as a subset of the natural numbers, i.e. $\texttt{A.field1} \subseteq \mathbb{N}$, the *encapsulation* transformation can be applied, which replaces the loop over all table field values with a loop over a subset of the natural numbers.

With the encapsulation transformation, a loop

```
forelem (ii; ii ∈ A.field1)
```

where $\texttt{A.field1} = \{1, 2, 6, 7, 8, 10\}$, is replaced with:

```
forelem (ii; ii ∈ ℕ10)
```

with $\mathbb{N}_{10} = [1..10]$. In the encapsulated loop, the values $3, 4, 5, 9$ will be iterated, but note that no tuple will exist where `field1` equals any of these values. As a result, the inner loop is not executed for these values, maintaining the iteration space of the original loop.

6 Materialization

In this section, the materialization transformation is described, which materializes the tuples iterated by a *forelem* loop using the accompanying index set to an array in which the data is represented in consecutive order and is accessed with integer subscripts. Although this can be seen as a simple normalization operation, it is an important enabling step that allows the compiler to address and modify the order of data access to these arrays. In fact, by materialization the execution order of an inner loop is fixed. (In the case of nested loops, orthogonalization fixes the order of the outermost loop). After two forms of materialization have been introduced, a number of transformations targeting the order in which data access takes place will be described.

A distinction is made between loop-independent and loop-dependent materialization. In loop-independent materialization, conditions in the index set of the loop to be materialized are not dependent on one of the outer loops. Materialization will result in a one-dimensional array. In loop-dependent materialization, the resulting array will get an additional dimension for each dependent loop. Both cases of materialization will now be discussed in turn.

6.1 Loop Independent Materialization

We first consider loop-independent materialization. The following loop iterates all tuples of `A` whose field equals a value `X`:

```
forelem (i; i ∈ pA.field[X])
  ... A[i] ...
```

To be able to determine which tuples of `A` to access, the index set is used. This is, in fact, an indirection level. This indirection can be removed by materializing the index into the tuple space as an array `PA` which only contains the entries of `A` that should be visited by this loop. This results in:

```
forelem (i; i ∈ ℕ*)
  ... PA[i] ...
```

with $\mathbb{N}* = [0, |\texttt{PA}|)$. The array `PA` only contains elements from `A` for which the condition `A[i].field == X` holds. The compiler is now enabled to address the order in which the data in `PA` is accessed, while the execution order of the loop is not specified. For example, using the transformations that can be applied on the materialization form, which are described below, the compiler can determine to put entries in `PA` in a specific order. The loop control is selected at the concretization stage, where the compiler can ensure the loop control for the loop will iterate the items of `PA` consecutively.

For the general definition of loop-independent materialization, consider a loop iterating a sparse structure `A`:

```
forelem (i; i ∈ pA)
  ...  A[i]  ...
```

which is transformed to:

```
forelem (i; i ∈ ℕ*)
  ...  PA[i]  ...
```

with $\mathbb{N}* = [0, |\texttt{PA}|)$. This transformation materializes the sparse structure **A** to an one-dimensional array **PA**.

The transformation can also be applied if the loop to be materialized is nested in another *forelem* loop and the posed condition in the index set of the loop to be materialized is *independent* of the outer loop. Consider, for example, where the outer loop could be the result of the application of the encapsulation transformation:

```
forelem (i; i ∈ ℕn)
  forelem (j; j ∈ pA.field[X])
    ... A[j] ... B[i] ...
```

Materialization of the inner loop will enable the compiler to address the order of data access of **A** together with the other array or tuple space references. Materialization of the inner loop proceeds as explained above and the outer loop is untouched:

```
forelem (i; i ∈ ℕn)
  forelem (j; j ∈ ℕ*)
    ... PA[j] ... B[i] ...
```

with $\mathbb{N}* = [1, |\texttt{PA}|]$ and **PA** only containing items that satisfy the condition.

6.2 Loop Dependent Materialization

If a loop to be materialized is contained in a loop nest and the conditions of its index set have a dependency on another loop, then the above described loop-independent materialization cannot be applied. Instead, loop-dependent materialization must be used, which is described in this subsection. Because loop-dependent materialization will result in higher-dimensional arrays, this results in more opportunities for the compiler to address and modify the order of data access to these arrays.

In general, a loop-dependent materialization has the form:

```
forelem (i; i ∈ ℕo)
  ...
    forelem (n; n ∈ ℕt)
      forelem (p; p ∈ pA.(fieldi, ...,fieldn)[(i,...,n)])
         ... A[p] ...
```

The index set iterated in the inner loop has a dependency on one or more of the outer loops. The iteration of **A** is materialized to an iteration of a multi-dimensional array **PA**, in which each loop-dependent condition is represented as an additional dimension in **PA**. The array **PA** only contains these items that are iterated by the original index set on **A**:

```
forelem (i; i ∈ ℕ_o)
  ...
    forelem (n; n ∈ ℕ_t)
      forelem (p; p ∈ ℕ*)
        ... PA[i]...[n][p] ...
```

with $\mathbb{N}* = [0, |\texttt{PA}[\texttt{i}]...[\texttt{n}]|)$. After this transformation, `PA` only contains entries that satisfy the conditions of the original index set. The dimensions of the materialized array correspond with the original conditions and thus with the loops on which the condition depended. Loop transformations, such as Loop Interchange, will thus have an effect on the order in which the data of `PA` is accessed. By taking this into account, the compiler can determine an efficient order in which to store the elements of `PA`, which has at this point not been set in stone.

To illustrate the loop-dependent materialization, consider a simple nested loop:

```
forelem (i; i ∈ ℕ_n)
  forelem (j; j ∈ pA.row[i])
    ... A[j] ...
```

The index set of the inner loop, `pA.row[i]` is dependent on iterator `i` of the outer loop. As a consequence, the array `PA` will obtain a dimension for this iterator `i`. The result of the materialization transformation is as follows:

```
forelem (i; i ∈ ℕ_n)
  forelem (j; j ∈ ℕ*)
    ... PA[i][j] ...
```

with $\mathbb{N}* = [0, |\texttt{PA}[\texttt{i}]|)$. Because `i` was determining which row of `A` was iterated, in the transformed loop `i` still controls the order in which the rows of the original matrix `A` are accessed in the materialization `PA`.

In case the index set has dependencies on two loops, a three-dimensional array is generated. Naturally, this has more degrees of freedom for optimization than the two-dimensional materialization. The application of the transformation is similar in case of doubly-nested loops. In this example, the index set has dependencies on two different outer loops:

```
forelem (i; i ∈ ℕ_n)
  forelem (j; j ∈ ℕ_m)
    forelem (k; k ∈ pA.(row,col)[(i,j)])
      A[k].value = ...
```

This results in a three-dimensional array `PA`:

```
forelem (i; i ∈ ℕ_n)
  forelem (j; j ∈ ℕ_m)
    forelem (k; k ∈ ℕ*)
      PA[i][j][k].value = ...
```

with $\mathbb{N}* = [0, |\texttt{PA}[\texttt{i}][\texttt{j}]|)$.

6.3 Transformations on the Materialized Form

After a *forelem* loop has been put in a materialized form, the data to be processed has been put in an array in consecutive order and is accessed with integer subscripts. At this stage, the compiler can modify the exact order of data access to these arrays and how this data is stored. In this section, a number of transformations are described that affect the storage of the data processed by a loop nest.

Horizontal Iteration Space Reduction The aim of Horizontal Iteration Space Reduction is to reduce unused fields from a table's schema. In fact, it is possible to perform this transformation before the materialization stage.

Formally, the transformation is defined as follows. Let `T` be a table with schema $\mathscr{S}(\texttt{T}) = (\texttt{field1 field2 field3 field4})$, `C` a list of condition fields $\texttt{C} \subset (\texttt{field1 field2})$ and `V` a list of values. Consider the loop nest:

```
forelem (k; k ∈ pT.C[V])
  ℛ = ℛ ∪ T[k].field1 + T[k].field2
```

We define a new table $\texttt{T}' \subseteq \texttt{T}$ with $\mathscr{S}(\texttt{T}') = (\texttt{field1 field2})$ and replace the use of `T` with `T'` in the loop.

Structure splitting Before materialization, tables are represented as multisets of tuples, accessible with integer subscripts. By default, the array that is the result of the materialization operation is an array of tuples, or structures. In some cases, it is more efficient to use a structure of arrays, i.e. the structures are split [15,7]. Within the *forelem* framework, this is defined as the *structure splitting* transformation.

Consider the materialized loop nest:

```
forelem (i; i ∈ ℕ_m)
  forelem (k; k ∈ ℕ*)
    ... PA[i][k].value ...
```

Structure splitting will modify the data storage of the array and convert the data accesses in the loop to:

```
forelem (i; i ∈ ℕ_m)
  forelem (k; k ∈ ℕ*)
    ... PA.value[i][k] ...
```

$\mathbb{N}*$ materialization Materialized loops use the $\mathbb{N}*$ index set as the set of integer subscripts to access the materialized array. How exactly these integer subscripts are stored is initially encapsulated within $\mathbb{N}*$ and can be made explicit using $\mathbb{N}*$ materialization.

Consider the following loop, the result of a materialization to `PA`:

```
forelem (i; i ∈ ℕ_m)
  forelem (k; k ∈ ℕ*)
    ... PA[i][k] ...
```

As a prerequisite for the final code generation stage, $\mathbb{N}*$ must be made explicit. This can be achieved by converting $\mathbb{N}*$ to a set `PA_len`. There are different means in which this set can be defined. The first is to define the set as follows:

```
PA_len[q] = max(len(PA[q]))
```

in which case all `PA_len[q]` values are the same and a single set containing integers up to the maximum value can be stored for this loop nest. Padding is inserted in the array `PA` for the values `PA[i][k]` with `k >= PA_len[i]`. The second means to create this array is to avoid inserting padding in `PA`. In this case `PA_len[q] = len(PA[i])`.

Regardless of which means is chosen, the resulting loop after $\mathbb{N}*$ materialization is:

```
forelem (i; i ∈ ℕ_m)
  forelem (k; k ∈ PA_len[i])
    ... PA[i][k] ...
```

Note that, in this loop the iteration order is still undefined. Only $\mathbb{N}* = [0, \mathbb{N}*)$ has been replaced with $\texttt{PA\_len}[\texttt{i}] = [0, \texttt{PA\_len}[\texttt{i}])$. In a subsequent concretization step the iteration order will be determined. For example, the loop:

```
  forelem (k; k ∈ PA_len[i])
```

is then concretized to:

```
   for (k = 0; k < PA_len[i]; k++)
```

$\mathbb{N}*$ sorting In case of loop-dependent materialization, $\mathbb{N}*$ encapsulates the sets of integer subscripts used for iteration of the inner loop. These sets are ordered irrespective of their cardinality. If the loop is to be parallelized, it is beneficial if the work is divided into blocks with evenly sized values for `PA_len` (after $\mathbb{N}*$ materialization). One way to achieve this is by imposing an order on the iteration of $\mathbb{N}*$.

The aim of $\mathbb{N}*$ sorting is to find an order of the iterator values `i` such that the value of $\mathbb{N}*$ decreases with subsequent iterations of the outer loop on `i`

```
forelem (i; i ∈ ℕ_m)
  forelem (k; k ∈ ℕ*)
    ... PA[i][k] ...
```

Consider that $\mathbb{N}* = [0, \texttt{len}(\texttt{PA}[\texttt{i}]))$. The goal is to iterate through $\mathbb{N}_m$, such that `len(PA[i])` decreases. Let $\texttt{perm}(\mathbb{N}_m)$ store the permutation of $\mathbb{N}_m$ for which this holds. Then, the loop is transformed to:

```
forelem (i; i ∈ perm(ℕ_m))
  forelem (k; k ∈ ℕ*)
    ... PA[i][k] ...
```

Note that this will affect the order of the data **PA**, which will be put in the corresponding sorted order at the concretization stage.

Dimensionality Reduction Loop-dependent materialization results in a multi-dimensional array by default. If this array is concretized as a multi-dimensional array, padding may have to be inserted for the uneven lengths of the rows. It is possible to avoid the introduction of this padding by storing the rows back to back. This reduces the dimensionality of the materialized array.

Consider the loop nest:

```
forelem (i; i ∈ ℕ_m)
  for (k = 0; k < PA_len[i]; k++)
    ... PA[i][k] ...
```

to reduce the dimensionality of the materialized array **PA** by one, this is transformed into:

```
forelem (i; i ∈ ℕ_m)
  for (k = PA_ptr[i]; k < PA_ptr[i+1]; k++)
    ... PA[k] ...
```

Based on the **PA_len** array, a new **PA_ptr** array is introduced, which keeps track of the start and end of each row in **PA**. Note that, the order of the iteration domain $[\texttt{PA\_ptr}[\texttt{i}], \texttt{PA\_ptr}[\texttt{i}+1])$ does not have to be defined and could be in any order.

7 Concretization

In Concretization, a *forelem* loop iterating a subset of integers is transformed into a regular *for* loop. This implies that a specific iteration order of the subset of integers is chosen. As example, consider the following loop, which is the result of a materialization transformation:

```
forelem (i; i ∈ ℕ*)
  ... PC[i] ...
```

First, $\mathbb{N}*$ materialization is applied, resulting in:

```
forelem (i; i ∈ PA_len)
  ... PC[i] ...
```

then the loop can be subsequently concretized to:

```
for (i = 0; i < PA_len; i++)
  ... PC[i] ...
```

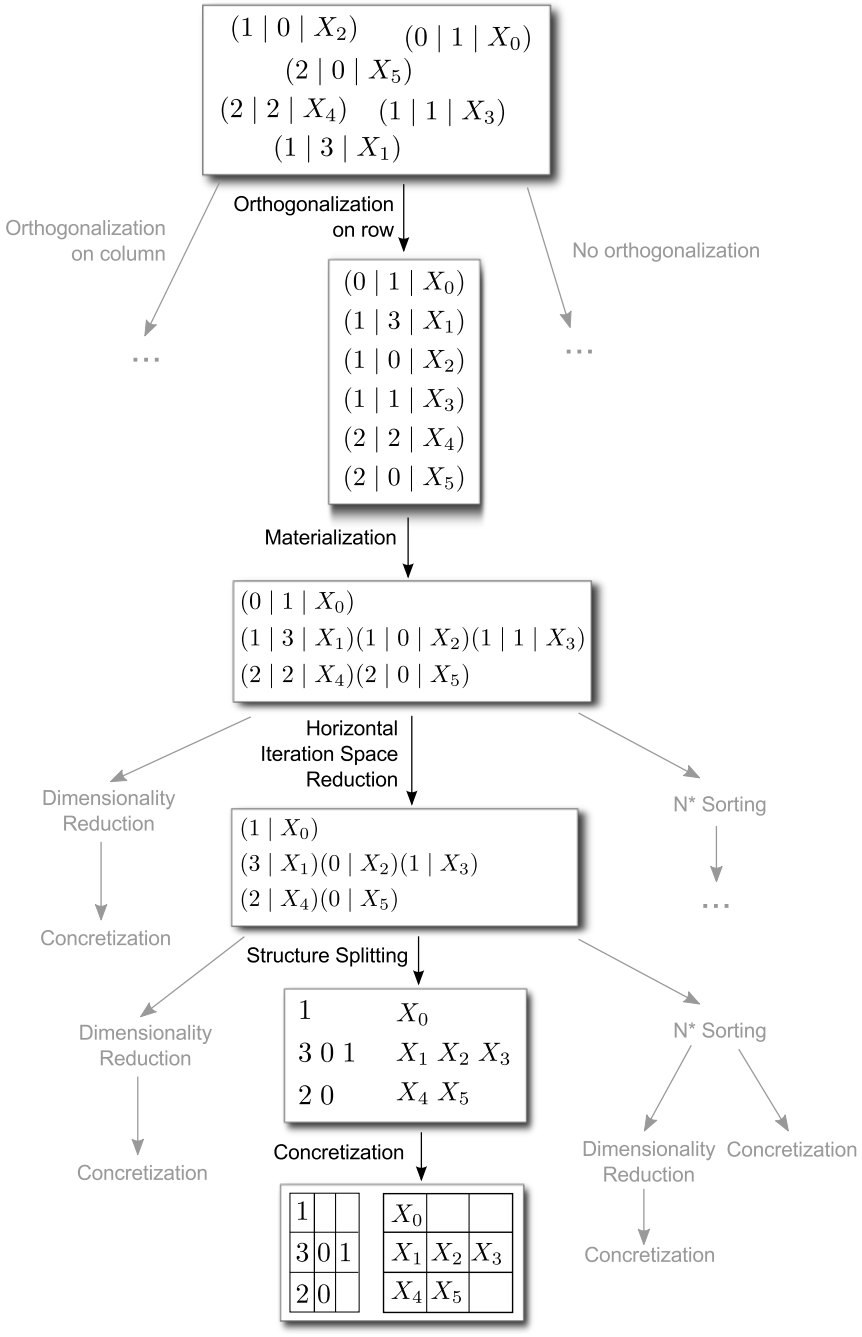

Fig. 1. An illustration of the application of orthogonalization, materialization and concretization on sparse matrix tuples in (`row` | `col` | `value`) format. The result of this concretization is commonly known as the ITPACK format (the arrays are stored in row-major order). The arrows displayed in gray depict a non-exhaustive set of other possibilities.

Concretization is a simple one-to-one mapping from the given materialized loop to a C *for* loop that can be compiled by a regular C compiler. Essentially, at this point the data storage format is generated that has been chosen by the optimization process. Using the different transformations that can applied on a materialized loop, described in the preceding section, many different storage formats can be generated for a single loop nest.

In the context of sparse matrix computations, sparse matrices can be represented as tables by storing the nonzeroes of the sparse matrix as tuples. The computation is expressed as a *forelem* loop operating on this table. Through the described transformations, orthogonalization, materialization and concretization, many different loops and accompanying data storage formats can be generated that achieve the same result. Figure 1 illustrates how such data formats are derived, starting from an unordered set of tuples. Established data storage formats, such as ITPACK and Jagged Diagonal Storage format [4], simply follow from the application of the transformations described in this paper. For example, the transformation sequence drawn in black in Figure 1 results in ITPACK format (considering row-major order storage of arrays in memory). When the structure splitting transformation is followed by dimensionality reduction, Compressed Row Storage (CSR) format is generated. Similarly, a transformation sequence that continues from orthogonalization on column can result in Compressed Column Storage (CCS) format.

8 Implementation of the Forelem Framework

The *forelem* framework was initially envisioned to support the integral optimization of database applications. To be able to support different programming languages and database APIs, a generic library was designed: *libforelem*. This library is capable of creating and manipulating *forelem* loop nests, by representing these using an internal Abstract Syntax Tree (AST). Different applications can make use of *libforelem* to create and manipulate *forelem* ASTs. For example, to compile stand-alone SQL queries to code, *libforelem* is used from a simple wrapper program that inputs the query and table schemas, transforms the query into *forelem* AST, and invokes a code generator of choice to generate the code. Other parsers that take a certain language as input and produce *forelem* loops can be developed next to the SQL parser, so that other problem domains can be supported.

The *libforelem* library is capable of performing various analyses and transformations on the *forelem* AST. Many of the implemented transformations are inspired by standard compiler (loop) transformations. An abstract code generation interface is present in the library to generate code from any *forelem* AST. Currently, the output of C/C++ code and algebraic *forelem* is supported. However, the use of *forelem* loops is not restricted to C/C++ and other languages can be supported by implementing the abstract code generation interface.

For our approach of vertical integration of database applications that is described in the Introduction, *libforelem* is used from a prototype Clang [1] com-

piler plugin. This plugin scans a C/C++ AST for calls to database API and extracts the performed operations, such as exact query strings that are requested to be executed. The extracted information is passed to *libforelem*. Transformations can then be performed as an interplay between the C/C++ AST and the *forelem* AST created by *libforelem*. Finally, code in the C/C++ source code is replaced with code generated using *libforelem*.

Note, that *forelem* loops are only used by the compiler tooling and are never visible to the end user. The general nature of the *forelem* framework allows for its usage with other problems. For example, to support the expression of BLAS kernels in the *forelem* AST, a C/C++ compiler is extended to be able to handle the *forelem* algebraic representation. A modified compiler can parse this representation into the *forelem* AST, using the *libforelem* library. Subsequently, the *forelem* AST can be manipulated in tandem with transformations on the regular C/C++ code. As a final step, C/C++ code is generated from the *forelem* AST and inserted in place of the *forelem* algebraic expressions, after which native code is generated by the regular C/C++ compiler

9 Conclusions

In this paper, we have described the *forelem* framework: a versatile unifying optimization platform for tuple-based computations. An overview has been presented of all transformations that are included within the *forelem* framework up till now. These include transformations that target the loop structure and a chain of transformations, from orthogonalization to concretization, that is capable of deriving efficient data storage formats for a given loop nest.

There are many areas in which the *forelem* framework can be put to use. Currently, the *forelem* framework is capable of generating query codes that are capable of achieving the same level of performance as these from state-of-the-art database systems. Other areas in which the *forelem* framework has been used are the vertical integration of database applications, optimization of sparse matrix kernels and Big Data applications. We plan to further improve the optimization strategies and code generation capabilities of the *forelem* framework as a future work. In forthcoming work, we also plan to address other application areas, such as graph computations, and study the relationship of the *forelem* framework with the Linda model [6].

References

1. "clang" C Language Family Frontend for LLVM. http://clang.llvm.org/
2. Allen, F.E., Cocke, J.: A program data flow analysis procedure. Commun. ACM 19(3), 137– (Mar 1976)
3. Allen, R., Kennedy, K.: Automatic translation of fortran programs to vector form. ACM Trans. Program. Lang. Syst. 9, 491–542 (October 1987)
4. Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., Van Der Vorst, H.: Templates for the solution of algebraic eigenvalue problems: a practical guide, vol. 11. Society for Industrial Mathematics (1987)

5. Bull, J.: Feedback guided dynamic loop scheduling: Algorithms and experiments. In: Euro-Par'98 Parallel Processing. pp. 377–382. Springer (1998)
6. Carriero, N.J., Gelernter, D., Mattson, T.G., Sherman, A.H.: The linda alternative to message-passing systems. Parallel computing 20(4), 633–655 (1994)
7. Curial, S., Zhao, P., Amaral, J.N., Gao, Y., Cui, S., Silvera, R., Archambault, R.: MPADS: memory-pooling-assisted data splitting. In: ISMM '08: Proceedings of the 7th international symposium on Memory management. pp. 101–110. ACM, New York, NY, USA (2008)
8. Kennedy, K.: A survey of data flow analysis techniques, pp. 5–54. Prentice-Hall, Englewood Cliffs NJ (1981)
9. Kennedy, K., McKinley, K.: Maximizing loop parallelism and improving data locality via loop fusion and distribution. In: Banerjee, U., Gelernter, D., Nicolau, A., Padua, D. (eds.) Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, vol. 768, pp. 301–320. Springer Berlin / Heidelberg (1994)
10. Kuck, D.J., Kuhn, R.H., Padua, D.A., Leasure, B., Wolf e, M.: Dependence graphs and compiler optimizations. In: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 207–218. POPL '81, ACM, New York, NY, USA (1981)
11. Polychronopoulos, C.D., Kuck, D.J.: Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. Computers, IEEE Transactions on 100(12), 1425–1439 (1987)
12. R., A.J.: Dependence Analysis for Subscripted Variables and its Applicati ons to Program Transformations. PhD Dissertation, Rice University (1983)
13. Rietveld, K.F.D., Wijshoff, H.A.G.: Quantifying Energy Usage in Data Centers Through Instruction-Count Overhead. In: SMARTGREENS 2013, 2nd International Conference on Smart Grids and Green IT Systems (May 2013)
14. Rietveld, K.F.D., Wijshoff, H.A.G.: To Cache or Not To Cache: A Trade-off Analysis For Locally Cached Database Systems. In: ACM International Conference on Computing Frontiers (May 2013)
15. van der Spek, H.L.A., Groot, S., Bakker, E.M., Wijshoff, H.A.G.: A compile/runtime environment for the automatic transformation of linked list data structures. Int. J. Parallel Program. 36(6), 592–623 (Dec 2008)
16. Tzen, T.H., Ni, L.M.: Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. Parallel and Distributed Systems, IEEE Transactions on 4(1), 87–98 (1993)
17. Van Der Spek, H.L.A., Wijshoff, H.A.G.: Sublimation: expanding data structures to enable data instance specific optimizations. In: Proceedings of the 23rd international conference on Languages and compilers for parallel computing. pp. 106–120. LCPC'10, Springer-Verlag, Berlin, Heidelberg (2011)
18. Zima, H., Chapman, B.: Supercompilers for parallel and vector computers. ACM, New York, NY, USA (1991)