Data-Centric Transformations on Non-Integer Iteration Spaces

Swarup Kumar Sahoo  Gagan Agrawal
Department of Computer Science and Engineering
Ohio State University, Columbus OH 43210
{sahoo,agrawal}@cse ohio-state.edu

ABSTRACT
Data-centric transformations have been used in recent years to improve locality for several classes of applications. However, the existing work has applied these transformations for integer iteration spaces, i.e., the iteration spaces involving loop variables that take integer values between specified lower and upper bounds. In many applications, a loop could involve a loop variable which takes values from a sequence or set of real numbers, strings, or any other data type. We refer to such iteration spaces as non-integer iteration spaces.

This paper focuses on the problem of applying data-centric transformations on applications with non-integer iteration spaces. We first present a general algorithm that uses a hash table. Then, we show how in many cases, we can exploit the repetitive nature of the dataset to avoid the overhead associated with such a table. Our algorithms have been implemented as part of a compiler for the XML query language XQuery, which supports processing over virtual XML. Our system also parallelizes the processing. We present experimental results from several application to demonstrate the effectiveness of our transformations and parallel performance.

1. INTRODUCTION
Declarative and/or application-class specific languages are often successful in easing application development. One example of such a language that has been widely used is MATLAB. Several projects over the last decade have demonstrated that compiler technology can be used for achieving high-performance starting from MATLAB codes [23, 19].

Such languages usually simplify application development through the use of high-level abstractions. However, such abstractions can also create difficulty in applying certain classes of restructuring transformations. Loop transformations and other transformations for improving locality have usually been developed and implemented in the context of imperative languages like Fortran and C.

In this paper, we focus on a class of transformations referred to as data-centric transformations [8, 14]. The goal of these transformations is to transform the code so that input data is brought into the memory or cache in chunks or shackles. After a chunk is read into memory, the program fragments or iterations of a loop which need this data are executed. If a program can be executed in this fashion without redundant data accesses, efficiency is achieved because of improved data access locality.

The existing work in this area has been applied on cases when iteration spaces involve one or more loop variables taking integer values between a specified lower-bound and upper-bound. In such cases, we say that the iteration space is integer-based. In this paper, we focus on cases when a loop involves a loop variable which takes values from a sequence or set of real numbers, strings, or any other data type. Such loops can be easily expressed in declarative and object-oriented languages. We consider the iteration spaces for such loops as being non-integer based. The existing algorithms for data-centric transformations do not apply to such cases.

This paper makes the following contributions. After introducing the notion of data-centric transformations on non-integer iteration spaces, we present a general algorithm that uses a hash table. Then, we show how in many cases, we can exploit the repetitive nature of the dataset to avoid the overhead associated with such a table. Our algorithms have been implemented as part of a compiler for the XML query language XQuery, which supports processing over virtual XML. Our system also parallelizes the processing. We present experimental results from several application to demonstrate the effectiveness of our transformations and parallel performance.

The rest of this paper is organized as follows. Section 2 gives background information on XML, XML Schemas, and XQuery. Overview of the problem and the context in which our work has been done are described in Section 3. Details of the transformation algorithms are presented in Section 4. Experimental results are presented in Section 5. We compare our work with related research efforts in Section 6 and conclude in Section 7.

2. BACKGROUND: XML, XML SCHEMAS, AND XQUERY
This section gives background on XML, XML Schemas, and XQuery.

2.1 XML and XML Schemas
XML provided a simple and general facility which is useful for data interchange. Though the initial development of XML was mostly for representing structured and semi-structured data on the web, XML is rapidly emerging as a general medium for exchanging information between organizations. XML and related technologies form the core of the web-services model [9] and the Open Grid Services Architecture (OGSA) [10].

XML models data as a tree of elements. Arbitrary depth and width is allowed in such a tree, which facilitates storage of deeply nested data structures, as well as large collections of records or structures. Each element contains character data and can have attributes composed of name-value pairs. An XML document represents elements, attributes, character data, and the relationship between them by simply using angle brackets.

Applications that operate on XML data often need guarantees on the structure and content of data. XML Schema proposals [1, 3] give facilities for describing the structure and constraining the contents of XML documents. The example in Figure (a) shows an XML document containing records of students. The XML Schema
describing the XML document is shown in Figure (b). For each student tuple in the XML file, it contains two string elements to specify the last and first names, one date element to specify the date of birth, and one element of float type for the student’s GPA.

### 2.2 XML Query Language: XQuery

As stated previously, XQuery is a language recently developed by the World Wide Web Consortium (W3C). It is designed to be a language in which queries are concise and easily understood, and to be flexible enough to query a broad spectrum of information sources, including both databases and documents.

XQuery is a functional language. The basic building block is an expression. Several types of expressions are possible. The two types of expressions important for our discussion are:

- **FLWR expressions**, which support iteration and binding of variables to intermediate results. FLWR stands for the keywords **for**, **let**, **where**, and **return**.
- **Unordered expressions**, which use the keyword **unordered**. The unordered expression takes any sequence of items as its argument, and returns the same sequence of items in a non-deterministic order.

We illustrate the XQuery language and the **for**, **let**, **where**, and **return** expressions by an example, shown in Figure 2. In this example, two XML documents, `depts.xml` and `emps.xml` are processed to create a new document, which lists all departments with ten or more employees, and also lists the average salary of employees in each such department.

In XQuery, a **for** clause contains one or more variables, each with an associated expression. The simplest form of **for** expression, such as the one used in the example here, contains only one variable and an associated expression. The evaluation of the expression typically results in a sequence. The **for** clause results in a loop being executed, in which the variable is bound to each item from the resulting sequence in turn. In our example, the sequence of distinct department numbers is created from the document `depts.xml`, and the loop iterates over each distinct department number.

```xml
<student>
  <firstname>Darin</firstname>
  <lastname>Sundstrom</lastname>
  <DOB>1974-01-06</DOB>
  <GPA>3.73</GPA>
</student>

(a) XML example

Schema Declaration

```xml
<xs:element name="student">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
      <xs:element name="DOB" type="xs:date"/>
      <xs:element name="GPA" type="xs:float"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

(b) XML Schema
```

![Figure 1: XML and XML Schema](image)

```xml
Figure 2: An Example Using XQuery's FLWR and Unordered Expressions
```

A **let** clause also contains one or more variables, each with an associated expression. However, each variable is bound to the result of the associated expression, without iteration. In our example, the **let** expression results in the variable `$e` being bound to the set or sequence of employees that belong to the department `$d`. The subsequent operations on `$e` apply to such sequence. For example, `count($e)` determines the length of this sequence.

A **where** clause serves as a filter for the tuples of variable bindings generated by the **for** and **let** clauses. The expression is evaluated once for each of these tuples. If the resulting value is true, the tuple is retained, otherwise, it is discarded. A **return** clause is used to create an XML record after processing one iteration of the **for** loop. The details of the syntax are not important for our presentation.

The last keyword we explain is **unordered**. By enclosing the **for** loop inside the **unordered** expression, we are not enforcing any order on the execution of the iterations in the **for** loop, and in generation of the results. Without the use of **unordered**, the departments need to be processed in the order in which they occur in the document `depts.xml`. However, when **unordered** is used, the system is allowed to choose the order in which they are processed, or even process the query in parallel.

3. **DATA CENTRIC TRANSFORMATIONS**

In this section, we describe that class of transformations we consider as data-centric transformations. Then, we introduce the problem of data-centric transformations on **non-integer** iteration spaces. Finally, we describe the context in which we have developed our algorithms.

3.1 **Overview of transformations**

**Data Centric Transformations** [8, 14] are a general class of transformations where data input to a program is brought into the memory or cache in **chunks** or **shackles**. After a chunk is read into memory, the program fragments or iterations of a loop which need this data are executed. If a program can be executed in this fashion without redundant data accesses, efficiency is achieved because of improved data access locality.

We use Oil Reservoir Management application [24] as an example for illustrating this class of transformations. The Oil Reservoir Management is an application to support cost-effective and environmentally safer production of Oil from reservoirs [24]. Using complex numerical models, simulations are carried out on a three-dimensional grid. At each time step, the value of 17 variables and cell locations in 3-dimensional space are output for each cell in the grid. Analysis of such simulation data can provide very useful in-
unordered
for $i$ in ($x_1$ to $x_2$)
for $j$ in ($y_1$ to $y_2$)
for $k$ in ($z_1$ to $z_2$)
let $Sp :=$ document("OilRes.xml")//data[$(x=$i) and $(y=$j) and $(z=$k) and $(time \geq \text{tmin})$ and $(time \leq \text{tmax})$]
return
<info>
  <x-coord> $i$ </x-coord>
  <y-coord> $j$ </y-coord>
  <z-coord> $k$ </z-coord>
</info>

define function analyze (element data $p$ )
as boolean
{
if (empty($p$))
then true
else
let $result :=$ analyze (subsequence($p$,2))
let $q :=$ item-at($p$,1)
return
if ($q/velocity > 0.7$) and ($q/mom-x * q/mom-x + q/mom-y * q/mom-y + q/mom-z * q/mom-z < 50.0$)
then $result$
else false
}

Figure 3: Oil Reservoir Data Analysis Using XQuery

The data processed by this application can be thought as being a simple collection of tuples, where each tuple corresponds to one grid-cell for one time-step. The $x$, $y$, $z$, and time values, denoting coordinates and time respectively, are stored explicitly along with other attributes whose value is generated. For simplicity, the attributes we consider are $velocity$, $mom - x$, $mom - y$ and $mom - z$.

An example XQuery which processes this data is shown in Figure 3. This query is based on the notion of bypassed oil cells or regions. An expression involving several attributes is used to determine if a grid cell is bypassed for a particular time-step. The query we consider specifies a spatial region and a range of time-steps, and requires the computation of grid cells within that spatial region that are bypassed for every time-step within the given range. The code iterates over the three-dimensional space for which the output is desired. Since the order in which the points are processed is not important, we use the directive unordered. Within an iteration of the nested for loop, the let statement is used to create a sequence of all cells that correspond to the particular spatial coordinates and the range of time values. The query involves determining if the defined condition evaluates to true for all the time periods within the specified time interval.

It is easy to see that without aggressive transformations, the execution of this query will be very inefficient. Each execution of the let expression will involve a complete scan over the dataset, since we need to find all data-elements that will belong to the sequence. However, we can recognize the computation in the recursive function is a reduction operation involving associative and commutative operators only. This implies that instead of creating a sequence and then applying the recursive function on it, we can initialize the output, process each element independently, and update the output using the identified associative and commutative operators. This recursion transformation facilitates application of Data Centric Transformation. The overall idea is to iterate over each data element and find out the iterations of the original loop in which they are accessed. Then, the computations corresponding to these iteration instances are executed. The details of the algorithms have been reported in previous work [8, 14, 16]. A key initial step in each of these algorithms is to identify the iteration spaces corresponding to the loops in the program.

3.2 Data Centric Transformations on Non-integer Iteration Spaces

let $Src =$ document("Oil.xml")//data/$x$/y$/z$
let $Coord =$ distinct-values ($Src$)
unordered
for $C$ in $Coord$
let $p :=$ document("OilRes.xml")//data[$(x=$C/x) and $(y=$C/y) and $(z=$C/z) and $(time \geq \text{tmin})$ and $(time \leq \text{tmax})$]
where ($p/x = \text{Sp}/x$) and ($p/y = \text{Sp}/y$) and ($p/z = \text{Sp}/z$)
return
<info>
  <x-coord> $[C/x]$ </x-coord>
  <y-coord> $[C/y]$ </y-coord>
  <z-coord> $[C/z]$ </z-coord>
</info>

define function analyze (element data $p$ )
as boolean
{
if (empty($p$))
then true
else
let $result :=$ analyze (subsequence($p$,2))
let $q :=$ item-at($p$,1)
return
if ($q/velocity > 0.7$) and ($q/mom-x * q/mom-x + q/mom-y * q/mom-y + q/mom-z * q/mom-z < 50.0$)
then $result$
else false
}

Figure 4: Modified Oil Reservoir Data Analysis Using XQuery

This paper mainly focuses on data centric transformations on loops when iteration spaces are not integer based. In most scientific codes written using imperative languages, iteration spaces involve one or more loop variables taking integer values between a specified lower-bound and upper-bound. However, in some cases, a loop could involve a loop variable which takes values from a sequence or set of real numbers, strings, or any other data type. Such loops can be easily expressed in XQuery, or other declarative or object-oriented languages.

We use a different (and more realistic) version of Oil Reservoir Management [24] and Transaction Database Analysis [27] to illustrate the difficulty in applying data centric transformation in such cases. The XQuery code for the modified version of Oil Reservoir Management is shown in Figure 4. In this case, the coordinates of each cell in the 3-dimensional space are real numbers, instead of integers. Hence, we cannot create an iteration space comprising of loops that iterate over integers within a specified range. Instead, a set of unique tuples, each denoting coordinates of one cell, needs to be created. Then, we iterate over this set. In each iteration the let statement creates a sequence of all cells that correspond to the
particular spatial coordinates and the range of time values. Finally, we apply the reduction computation on the obtained sequence.

Because the iteration space is not as well structured as in the previous example we considered, we cannot directly create the set of output elements corresponding to the iteration space. Thus, the existing algorithms cannot be directly applied to these codes.

```xml
<xs:schema
    xsi:namespace="http://example.org"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="largeItemset">
    <xs:sequence>
      <xs:element name="items" type="xs:unboundedSequence/"/>
      <xs:element name="support" type="xs:integer"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

**Figure 5: Transaction database Analysis Using XQuery**

Another example of this type of code is transaction database analysis. The part of the code we are interested in is shown in Figure 5. Each such transaction contains names of a set of items. The shown XQuery code creates the set of unique items from the input dataset. Then, it iterates over this set. In each iteration, the let statement creates a sequence of items with the same name as that of the particular item. Then, we apply operation count on the resulting sequence. Thus, the iteration space here corresponds to the set of unique items in the dataset. The existing algorithms cannot be directly applied here either.

3.3 Overview of Our Approach and System

Before discussing the algorithms, we describe the context in which our work is applied. We are considering XQuery over virtual XML data. By virtual XML data, we imply low-level data with a logical or virtual view based on a high-level XML Schema.

The motivation for supporting virtual XML is as follows. Scientific datasets are typically stored as binary or character flat-files. Because the underlying data is often multi-dimensional, chunked multi-dimensional layout [25] or standards like HDF5 [11] and NetCDF [22] are quite popular. Such low-level layouts enable compact storage and efficient processing. The use of low-level and specialized data formats, however, makes the specification of processing much harder.

The goal of our work is to alleviate the above problems. An overview of our system is shown in Figure 6. A high-level XML Schema is used as a virtual view of the dataset. Such Schemas are independent of the physical layout of dataset. An application developer will specify the desired processing using the XML query language XQuery [4]. The physical storage of the data is still be in compact and low-level formats.

A low-level XML Schema file is provided reflecting the actual physical layout and the metadata information. The mapping between each element of the high-level XML Schema and corresponding element in low-level XML Schema is described by a Mapping Schema. Our compiler generates C language code which executes on the low-level dataset.

We use the original and modified version of Oil Reservoir Management application [24], discussed earlier in this Section, for describing the notion of high-level, low-level, and mapping Schemas.

3.3.1 High Level Schema

The high-level XML Schema for the simulation dataset is shown in Figure 7. As a logical view, the data can be thought as being a simple collection of tuples, where each tuple corresponds to one grid-cell for one time-step. The x, y, z, and time values are stored explicitly along-with other attributes whose value is generated. For simplicity, the attributes we consider are velocity, mom – x, mom – y and mom – z. In addition, the High-Level schema for modified version of the application uses a special construct unique, which will be explained in Section 4.

3.3.2 Low level Schema

The simulation output is typically not stored in such a format as shown in high-level schema, as it will involve very high storage overheads. Instead, data is stored in a low-level compact format. We used HDF5, a hierarchical data format popular in many scientific communities as our low level data layout.

In our system, a low-level XML Schema file is provided to the compiler to specify the actual layout of the data in the low-level format. The low-level XML Schema for our example is shown in Figure 8. As we can see, the file coord stores the spatial coordinate values and the file info stores the variables associated with the spatial coordinates for different time-steps. The x, y, and z coordinates are stored as 3 separate components within the file coord. Each of these three components stores the coordinate values consecutively as one-dimensional array. The variables velocity and momentum along x, y, and z directions are stored as 4 separate components in the file info. Again, this data is stored consecutively as a one-dimensional array, where the index of one data point is the same as the index of its corresponding spatial coordinate. The information for all variables for a particular time-step is stored as one group, and there is a separate group for each time-step. Each group contains an attribute stating the time-step. Note that if each time step contains data for different set of cell locations, the coordinate values need to be stored separately for each time step in stead of storing them just once.

3.3.3 Mapping Schema

In our system, we also provide a Mapping Schema. This Schema describes the correspondence between high-level and low-level Schemas. The Mapping Schema for the Oil Reservoir Management example is shown in Figure 9.
The spatial coordinates corresponding to a data point in each component of a file can be computed using the Mapping Schema. According to the Mapping Schema, the velocity attribute of a tuple in high-level Schema is mapped to a data element of the one-dimensional array of the velocity component within the file, which is contained in the group data in the file info. The x coordinate for this data point in the high-level Schema is the value of dataset x, at the same offset as that of the corresponding data element of the velocity component, along the 1st dimension. The time value of this data element is the time value of the group data. We use a special construct index to specify the above features in the mapping Schema. In the mapping Schema,

\[ \text{Index}(/\text{low}/\text{info}/\text{data}/\text{velocity}, i) \]

implies the position of one data element of the velocity component, along the \(i^{th}\) dimension.

In our system, the low-level XML Schemas and mapping Schemas are expected to be invisible to the programmer writing the XQuery code. The goal is to provide a simplified view of the dataset to the application programmers, thereby easing the development of correct data processing applications. The compiler translating XQuery codes obviously has the access low-level XML Schemas and mapping Schemas, which enables it to generate efficient code.

4. COMPIILER ALGORITHMS

This section describes the various compiler algorithms we have developed.

4.1 Canonical Structure of the Source Code

We describe a canonical structure for the type of codes our transformation algorithms target. This structure is shown in Figure 10 and encompasses the XQuery codes we have targeted so far. The canonical structure is shown using the extended BNF notation. \{ \ldots \} implies that the expression is repeated zero or more times, [ \ldots ] implies that expression is optional, and | implies a choice between two expressions.

The first optional statement states that a list of tuples is created by projecting a set of variables from the document. The second optional statement creates a unique set of tuple from the created list. After these optional statements, the code comprises several codes obviously has the access low-level XML Schemas and mapping Schemas, which enables it to generate efficient code.

The iteration space is defined by the lists of tuples. In the second case, the iteration space is defined by sets of lower and upper bounds of integers. In each iteration, a sequence of tuples is created by the let statement where each tuple in the sequence satisfies a conditional expression. After that, a reduction computation involving an associative and commutative operator is applied on the obtained sequence. Finally, the result is returned.

4.2 Legality of the Transformation

We now describe how we determine the legality of the data-centric transformation.

Consider the canonical structure, and focus on the iteration space created by the for loops. There are no loop carried dependencies across these loops if we create separate variable sequences and return values in each iteration. Further, the unordered directive implies that the order of the return values in the output document is not important. Therefore, the iterations of these for loops can be executed in any order. If the reduction computation involves associative and commutative operators only, this computation can be applied on the sequence in any order. This means that iterations of the implicit loop involving the let expression can be reordered as well. Hence, the transformed code, in which we go through each data element in the document, and update the output element associated with the iteration accessing that data element, is legal.

Thus, the legality test involves checking that the code structure matches the canonical form, and identifying that the reduction computation involves only associative and commutative operators. The first part of the checking is relatively straightforward and an algorithm for performing the second test was described in our earlier work [16].

4.3 Algorithm Template

We now describe a general model for data-centric transformations and execution. This model is used for describing our algorithms in the next two subsections.

The overall idea is to iterate over each data element in the low-level data layout. We determine the iteration(s) of the original loop in which they are accessed. Then, the computations corresponding to these iteration instances are executed. Clearly, this idea needs to be extended to the cases when different datasets or portions of the file(s) need to be accessed together for performing the computations.

We create an iteration space corresponding to the high-level XQuery code. Let us suppose that a mapping function \( \mathcal{F} \) exists from the iteration space to the high-level Schema, denoting the data elements from the high-level Schema that are accessed in a given iteration. This can be computed from analysis of the XQuery code. Similarly, suppose the function \( \mathcal{C} \) gives the mapping from the data elements in
Our goal is to be able to compute the function $\mathcal{M}^{-1}$, which will map the data elements in the low-level Schema to the iterations. We may store a member named output in each hash table entry, denoting the output element corresponding to that particular iteration or create a separate array of output ele-

$\mathcal{M}_1^{-1}, \ldots, \mathcal{M}_n^{-1}$, respectively.

### 4.4 Standard Algorithm with Hash Tables

Now, let us revisit the modified oil reservoir management and transaction database analysis applications we had discussed earlier. In such applications, the iteration space is not integer based. The key difficulty with these kind of codes is that we cannot directly infer the actual iteration space or the output elements corresponding to each iteration.

The most direct, but possibly inefficient solution, to this problem involves the use of hash tables. We create an abstract integer iteration space, using the following procedure. The elements in the actual iteration space are stored in a certain order, for example, lexicographic. This provides a unique sequence number to each element. The abstract integer iteration space is created by using these sequence numbers to denote each iteration in the actual iteration space. Thus, there is a one-to-one correspondence between the two iteration spaces. We also create a set of output elements corresponding to the abstract iteration space.

The complete algorithm can be derived from the template discussed in the previous subsection, using these two steps. First, we need to construct the actual and abstract iteration spaces. Second, we need to create mapping from an iteration in the actual iteration space to the abstract iteration space. The second step can be efficiently implemented with the help of hash table data structure. The modified algorithm is shown in Figure 11. Here, we do not construct the abstract iteration space explicitly. Instead, we use the sequence number in the hash table as the iteration instance in the abstract iteration space. We may store a member named output in each hash table entry, denoting the output element corresponding to that particular iteration or create a separate array of output ele-

Figure 9: XML Mapping Schema for original and modified version of Oil Reservoir Simulation Data

```xml
[ let $SrcList = document("Document Name")//Set Of Variables ]
[ let $UniqueList = distinct-values($SrcList ) ]
unordered(
for $loopindex in $UniqueList{ for $loopindex in $UniqueList } )
let $sequence := $document("Document Name")//Set Of Variables
where ( Conditional-exp($sequence/Var, $loopindex ) )
{ and ( Conditional-exp($sequence/Var, $loopindex ) ) }
let $ReturnValue = ReductionComputation($sequence)
[ where Conditional-exp($ReturnValue ) ]
return

<Main-Tag>
<Var-Tag>{ expr($loopindex) } </Var-Tag>
{ Var-Tag } { expr($loopindex) } </Var-Tag>
</Main-Tag>
```
4.5 Efficient Code Generation Without Hash Tables

The algorithm using hash tables can be quite inefficient, especially when the size of the iteration space is large and hash table needs to be stored on a disk. In many cases, we do not need the step of creation of abstract iteration space, or the use of hash tables. This could be because the mapping from the actual iteration space to the abstract iteration space can be deduced automatically. In these cases, we go through the datasets and for each element, compute the iterations in the actual iteration space where they will be accessed. Then, we compute an expression which gives the corresponding iteration in the abstract iteration space. We apply the reduction computation and update the output element specified by the computed expression. In the modified version of Oil Reservoir Simulation, the actual iteration space consists of the set of cell locations. Suppose, we use the sequence number of the coordinate values, in the order in which they are stored, as the abstract iteration instance. Then, the mapping function can be very easily computed from the offset of the data elements. But, such mapping function can not be computed for the transaction database analysis application. In the rest of this subsection, we explain 1) the algorithm to detect when the hash table can be avoided, and 2) how we can apply the data centric transformation without using a hash table. Our algorithms particularly use the information about low-level layouts and the mapping between high-level and low-level layouts.

In the first step, we determine the procedure to create the actual iteration space from low-level datasets, using the XQuery source code and the mapping schema. Recall that the codes following our canonical structure project required tuples from datasets, and a unique sequence of tuples is formed from the collected tuples. Suppose, the required tuples consist of data elements $d_1, d_2, \ldots, d_n$ in the high-level schema. Let the corresponding mapping functions be $C_1, C_2, \ldots, C_n$. By analyzing these mapping functions and their inverse functions, we determine the datasets that will be involved in construction of actual iteration space and the procedure to construct it.

There are two options to consider in constructing the actual iteration space. In the first option, actual iteration space or the set of tuples is constructed by combining several datasets, such that each dataset forms one component of the tuple and then selecting unique tuples from this set. The modified version of the Oil reservoir simulation will use this option. In this case, each tuple in the set will have one component each from the datasets $x$, $y$, and $z$. The $x$, $y$, and $z$ data elements in the selected tuple will have the same...
offset in their original dataset. A unique sequence of tuples will then be selected from this set. In the second option, we create a set of tuples by combining multiple datasets as before, such that each dataset forms one component of the tuple. But, in this case, we will have several such sets and the actual iteration space is formed by taking union of all these sets and then selecting the unique tuples. Consider an alternative version of Oil Reservoir Simulation, in which each time step stores values for different set of cell locations. This version will use the second option. In this case, the low-level layout stores the x, y, and z coordinates separately for each time-step, along with the other attributes. A set of tuples will be formed for each time step and such a set will be formed for each different time step. All these sets need to be combined before selecting the unique tuples for constructing the actual iteration space. Similarly, the Transaction database analysis will also use the second choice. Mapping functions and low-level schema can be used directly to decide which of the two choices will be followed. From the previous analysis, we know which datasets need to be combined to form the set of tuples. If all these datasets are within a sequence statement in the low-level schema, implying that there are multiple instances of all these datasets, then the second choice will be followed. Otherwise, the first choice will be followed.

In the second step, after construction of the procedure to create abstract iteration space, we determine if we need a hash table for data centric transformation. This is based on which of the two choices above we are using, and also, if there are duplicate tuples in the constructed tuples from the low-level datasets. We use the unique clause in the high-level schema to detect the presence of duplicate tuples. Unique(d1, d2, ..., dn) clause indicates that there is no duplicate tuple (d1, d2, ..., dn) in the data.

Let us suppose that V is the set of datasets which form unique tuples and P is the set of datasets which will form tuples in the iteration space. In the modified version of the Oil reservoir simulation, V=(x,y,z,t) and P=(x,y,t).

If, P is not a subset of V, we conservatively conclude that duplicate tuples may be present in the set of tuples. In such a case, we use the hash-table based scheme. Next, we check if P and V are identical. If so, we can use the sequence number associated with data element accesses as the abstract iteration instance. Thus, hash table is not needed in this case.

Finally, we consider the possibility that P is a subset of V, but not identical to it. We take two different tuples p1 and p2 from P, and assume they have same value. We then use a fixed data element in V - P, along with p1 and p2, to construct two distinct data points d1 and d2. This is done by applying the mapping function C^-1. If it is possible to construct two such distinct data points, then there are no duplicate elements. Otherwise, duplicate elements may be present and we have to use hash table based scheme. As an example consider two different tuples (x1, y1, z1) and (x2, y2, z2) in the modified Oil reservoir simulation. From analysis of the mapping schema, we can infer that these two tuples can be associated with the same time value t to give two distinct tuples (x1, y1, z1, t) and (x2, y2, z2, t). Hence, we infer that there will be no duplicate tuples. In the case of first choice, the above test is sufficient to determine the presence of duplicate elements. But, in the case of second choice, p1 and p2 should be chosen once from the same set of tuples, and then from different set of tuples, and above test should be carried out for both the cases. These tests can all be performed using the inverse mapping function. Whenever there are no duplicate tuples, we do not need to use hash table. In all these cases we can use the sequence number or the offset associated with the data element accesses as the abstract iteration instance.

The summary of steps of the algorithm are shown in Figure 12 and the result of applying data centric transformation on the modified Oil Reservoir Simulation is shown in Figure 13.

5. EXPERIMENTAL RESULTS

We conducted two sets of experiments to evaluate our approach. In the first set, we evaluated the performance benefits of Data Centric Transformations using four applications, including comparing the versions with and without the use of hash table. In the second set, we evaluated the parallel performance for the transformed codes. The parallel experiments used parallel HDF5 version 1.6.3, which uses MPI-I/O for doing parallel I/O.

Our sequential experiments were carried out on a 700 MHz PIII machine with 1 GB memory and Linux version 2.4.18. We used four applications, which were, transaction database analysis, original and modified version of oil reservoir simulation, and virtual microscope [15]. We used 4000 items each with 20 items for transaction database analysis. Original oil reservoir simulation was run with a 20 x 20 x 20 grid with 100 different time-steps. The modified version used a 50 x 50 x 50 grid with 5 time-steps. The query range for both the applications was a 5 x 5 x 5 grid. The dataset for Virtual microscope involved 16 regions with 100 x 100 pixels, with 10 different resolutions and query range of 10 x 10 pixels.

We compared three different strategies for our set of applications. The first strategy uses data centric transformation without the hash table. This strategy could be correctly applied to only three of the four applications. The second strategy uses code with data centric transformation with a hash table. The third is a default strategy, where data-centric transformations are not used. Note that for the virtual microscope and the original oil reservoir simulation, each of which has integer based iteration spaces, we have not evaluate the second strategy.

The results from these experiments are presented in Figure 14. The performance improvement of versions with data centric transformation over the versions without data centric transformation varies between 8 and 12.5. Note that the datasets used here are small enough to be cached in memory. If larger datasets are used, we will expect even higher benefits from data-centric transformations. We can also observe that the version which does not use hash table is nearly 50% faster than the version which uses hash table for the modified oil reservoir simulation application. Thus, our algorithm for determining when hash tables are not needed results in substantial performance improvements.

Next, we report results from parallel experiments. These experiments were conducted on an Itanium 2 cluster at the Ohio Supercomputer Center. Each node in the cluster has two 1.3 GHz Intel
Our work on data-centric transformations derives from the original work in this area on dense numerical linear algebra codes by Pingali et al. [14]. In our earlier work, the focus has been on supporting such transformations on disk-resident datasets, and applying these transformations on XQuery codes written with high-level view of the datasets [8, 15]. In this paper, we have particularly focused on non-integer iteration spaces, and have developed new algorithms. Our work is also closely related to the work on automatically synthesizing sparse computations from dense codes [2, 18, 21]. These frameworks restructure codes to make the access patterns efficient for the storage format of the sparse matrix. These frameworks operate on two-dimensional sparse matrices, which can be stored in a limited number of well-structured formats. Our contribution is to show how XML Schemas can be systematically used by the compiler to support high-level abstractions and apply the transformations, without limiting how the low-level datasets can be organized. Our framework also does not require the low-level access functions to be implemented for all possible layouts. Also, to the best of our knowledge, there is no previous work on data-centric transformations on non-integer iteration spaces. Compiler optimizations for out-of-core data-structures have been considered by several projects [13, 20, 5]. This work has been done in the context of imperative languages, and non-integer iteration space have not been considered.

XQuery compilation is an active area of research in the database community. Significant attention has been paid to integrating and querying XML over relational databases [6, 26, 28, 17] and various approaches for supporting XQuery [12, 7]. We are not aware of any other efforts on supporting XQuery over scientific datasets, or considering the kind of transformations we have considered here.

7. CONCLUSIONS

This paper has focused on the problem of applying data-centric transformations on applications with non-integer iteration spaces. We have presented a general algorithm that uses a hash table. Then, we have shown how, in some cases, we can exploit the repetitive nature of a dataset to avoid the overhead associated with such a table.

Our algorithms have been implemented as part of a compiler for the XML query language XQuery, which supports processing over virtual XML. Our system also parallelizes the processing. We have presented experimental results from several application to demonstrate the effectiveness of our transformations and parallel performance.
8. REFERENCES


