An Approach for Code Generation in the Sparse Polyhedral Framework

Michelle Mills Strout[a,*], Alan LaMielle[b], Larry Carter[c], Jeanne Ferrante[c], Barbara Kreaseck[d], Catherine Olschanowsky[b]

[a] *Computer Science Department, University of Arizona, Tucson, AZ, USA*
[b] *Computer Science Department, Colorado State University, Fort Collins, CO, USA*
[c] *Computer Science & Engineering Department, University of California, San Diego, La Jolla, CA, USA*
[d] *Computer Science, La Sierra University, Riverside, CA, USA*

Abstract

Applications that manipulate sparse data structures contain memory reference patterns that are unknown at compile time due to indirect accesses such as `A[B[i]]`. To exploit parallelism and improve locality in such applications, prior work has developed a number of run-time reordering transformations (RTRTs). This paper presents the Sparse Polyhedral Framework (SPF) for specifying RTRTs and compositions thereof and algorithms for automatically generating efficient inspector and executor code to implement such transformations. Experimental results indicate that the performance of automatically generated inspectors and executors competes with the performance of hand-written ones when further optimization is done.

Keywords: inspector/executor strategies, runtime reordering transformations, sparse polyhedral framework

1. Introduction

Many scientific computing applications and virtually all graph algorithms use sparse data structures that are typically accessed using indirect array references such as `A[B[i]]`. Such applications are commonly called irregular applications, and examples include solving partial differential equations over irregular grids, molecular dynamics simulations, and sparse matrix computations. These computational simulations of physical phenomena are becoming increasingly important in the natural sciences. For example, molecular dynamics simulations are

[*] Corresponding author

Email addresses: `mstrout@cs.arizona.edu` (Michelle Mills Strout), `carter@cs.colostate.edu` (Larry Carter), `ferrante@cs.ucsd.edu` (Jeanne Ferrante), `kreaseck@gmail.com` (Barbara Kreaseck), `cathie@cs.colostate.edu` (Catherine Olschanowsky)

used to aid drug design and study protein interactions [1]. The performance of computational simulations is important because improved performance enables finer-grained modeling for a larger number of time steps.

Unfortunately, indirect array accesses often result in irregular memory reference patterns that exhibit poor locality and consequently can result in poor performance. Processors always move blocks of contiguous data into cache, so whenever a program references a single array element, the entire enclosing block is moved into cache. If the other elements of the block are used before the block is evicted, the program can often achieve acceptable performance. However, irregular memory references often do not have much localized reuse. In fact, a typical irregular application only achieves 5–10% of the advertised peak processor performance [2]. Poor data locality is becoming even more of a performance problem with multicore architectures where shared memory results in more cores competing for both space in cache and memory bandwidth; also, access to shared memory is becoming non-uniform.

There have been many program optimizations and transformation frameworks developed for improving the memory reference patterns for codes that are limited to affine references [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]. Currently, the dominant transformation framework for affine transformations is the polyhedral framework [14, 3, 5, 8, 15, 16, 17, 18]. There are two reasons these techniques cannot be applied when there are indirect memory references. The first is that indirect references inhibit the data dependence analysis needed to determine if a transformation preserves the semantics of the program. The second reason is more fundamental: it is usually impossible to know at compile time whether a particular indirect reference will lead to a good or bad access pattern — the access pattern depends on values in the index arrays that are only known at run-time.

To overcome these problems, Run-Time Reordering Transformations (RTRTs) have been developed [19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]. Typically, an RTRT is implemented using an *inspector* and an *executor*. The inspector is code that analyses the memory reference at runtime, perhaps by looping over the index array (the `B` array in `A[B[I]]`), to generate a new mapping for the data or a new order to execute the computation (e.g. by reordering entries in `B[]`) that improves the data locality or enhances parallelism. The executor is a modified version of the original code that incorporates the new data and computation orders. The inspector is called outside of a loop that calls the executor, so the time required by the inspector is amortized over many iterations of the executor. In this paper, we present the Sparse Polyhedral Framework (SPF) for the specification of computation with indirect memory references and program transformations on such computations, which are then implemented with generated inspectors and executors. The focus of this paper is code generation for data locality RTRTs.

Previous work has made some progress toward the automation of run-time reordering transformations. Initially, such transformations were incorporated into applications manually for parallelism [20]. Next, libraries with run-time transformation primitives were developed so that a programmer or compiler could insert calls to such primitives [31, 32]. Currently, there are run-time reordering transformations for which a compiler can automatically analyze and generate the inspectors [33, 24, 25, 30]. In general, theses techniques focus on individual inspector/executor strategies. Other than a small subset of "hard-coded" compositions, the generation of inspectors that implement a set of RTRTs has not been automated.

The components of a general automatic RTRT system should include:

1. A framework for specifying irregular computations and compositions of RTRTs to apply to these computations.
2. A library of RTRTs including compile-time and run-time support that can easily be applied to particular computations.
3. Program analysis algorithms that update information summarizing the effects of a sequence of RTRTs to determine when additional RTRTs are legal.
4. A guidance system to choose a sequence of RTRTs given various evaluation criteria such as minimizing execution time, maximizing throughput, and/or minimizing memory footprint.
5. A code generator capable of generating inspector and executor code.

Creating a complete automated system is beyond the scope of this paper. In particular, creating a good guidance system capable of automatically selecting effective program optimization strategies is a very challenging problem. Before one can automate the selection of a sequence of transformations, one must gain extensive experience with user-selected transformations. The contributions described in this paper aim at facilitating such experiments. In particular, we focus on goals (1), (2) and (5) listed above, leaving some of the analysis and all of the guidance to be provided by the experimenter.

In summary, the contributions of this paper are:

- A unified framework called the Sparse Polyhedral Framework (SPF) for specifying irregular/sparse computations and Run-Time Reordering Transformations (RTRTs) on such computations (goal 1).

- Description of a code generator prototype, called the Inspector/Executor Generator in Python (IEGen in Python), that enables the user to specify computations and transformations as a substitute for goals 3 and 4. The code generator fulfills goal 5 through the use of two new intermediate representations: (1) the Inspector Dependence Graph (IDG) to represent the components of a composed inspector and (2) the Mapping IR (MapIR) to represent the executor.

- Experimental results that explore how well our automatic generators compare against hand-coded and optimized inspectors and executors.

This paper is a more succinct version of a technical report [34] and includes updated experimental results.

Section 2 presents the Sparse Polyhedral Framework (SPF) and how the example transformations can be specified. Section 3 presents techniques for generating inspector and executor implementations from the Inspector Dependence Graph (IDG) and the Mapping Intermediate Representation (MapIR), and Section 4 describes how transformations can be implemented as manipulations of the IDG for the inspector and the MapIR for the executor. Section 5 evaluates the code generation techniques in terms of their performance in the context of a molecular dynamics benchmark and an sparse matrix vector product benchmark. Section 6 describes related work, and Section 7 concludes.

2. The Sparse Polyhedral Framework

RTRTs fall into two main classes. *Data reorderings* change the mapping of data to storage locations. They attempt to improve the spatial locality of the memory reference pattern, for instance, by placing values that will be referenced by nearby iterations in the same cache blocks. *Iteration reorderings* change the order that iterations of a loop (or loop nest) are executed. Here the goal might be to increase the temporal locality of iterations that access the same data. Often performance can be further improved by applying a sequence of RTRTs. A typical scenario is to first perform a data reordering, and follow it with an iteration reordering.

Specifying run-time data and iteration reorderings in a compile-time framework has several advantages. First, both run-time and compile-time transformations are uniformly described. Secondly, a framework supported with code generation algorithms enables experimenting with different compositions of existing RTRTs. Third, the sparse polyhedral framework enables the development and the eventual automatic selection of RTRT compositions. Finally, the transformation legality checks can provide constraints on the compile-time specification of RTRT compositions and on the run-time library of algorithms that generate run-time reordering functions.

In general, a transformation framework includes

- an intermediate representation for representing computations,
- transformation specifications,
- formalizations for applying transformations,
- formalizations for checking transformation legality, and
- algorithms for generating efficient code that implements the specified transformations.

Example frameworks include the unimodular transformation framework [35, 36] and various instances of the polyhedral framework [14, 3, 5, 8, 15, 16]. In the polyhedral framework, the static control parts (SCoP) [37, 38] of a program can been represented with some statement representation (e.g., an abstract syntax tree), an affine function for each memory accesses within each statement, affine functions to represent data dependences due to the memory accesses, and an affine scheduling function for each statement. Transformation specifications and data dependences are formalized as integer tuple functions. Transformations are performed within a polyhedral framework by applying affine transformation functions to the statement scheduling functions. Transformation legality checks can be performed by applying the transformation to the dependence abstraction and determining if the result is legal. Code generation algorithms generate code that will execute the transformed iteration space in lexicographical order.

This section reviews the Sparse Polyhedral Framework (SPF) for specifying irregular computations and Run-Time Reordering Transformations (RTRTs) on such computations. The SPF enables the explicit composition of run-time data and iteration-reordering transformations and was initially presented in [39]. As the name indicates, the Sparse Polyhedral Framework (SPF) is based heavily on polyhedral transformation frameworks, especially that of Kelly and Pugh [8]. Polyhedral frameworks focus on specifying transformations that can be completely specified and performed at compile time. The SPF enables the combined compile-time and run-time specification of run-time reordering transformations. Similar to the work in [40], the SPF uses uninterpreted function symbols such as $B(i)$ to represent non-affine memory references such as the indirect memory references `A[B[i]]`. Additionally, we can express run-time data and iteration-reordering within the SPF using uninterpreted function symbols.

2.1. Abstract Sets and Relations

Abstract sets and relations are the fundamental building blocks for the SPF. Data and iteration spaces are represented with abstract sets and access functions; transformations are represented with abstract relations. We use the term *abstract* to differentiate between sets and relations specified at compile time, which are abstract, and sets and relations that are explicitly constructed at runtime with all of their members, which are referred to as *explicit* sets and relations. This section defines abstract sets, abstract relations, and operations that can be performed on them.

Abstract sets are integer tuple sets with inequality and equality constraints on set membership,

$$\{[i_0, i_1, ..., i_{d-1}] \mid \text{ inequality and equality constraints } \}.$$

The *arity* of the set is the dimensionality of the tuples, which for the above is d. The constraints can be affine expressions of the tuple variables i_k, symbolic constants, existential variables, and uninterpreted function symbols.

Symbolic constants are computation parameters that do no change during the course of the computation. For example, the following set is a set of integer d-tuples parameterized by the symbolic constants N and B:

$$\{[i_0, i_1, ..., i_{d-1}] \mid (i_0 > 0) \wedge (i_0 < N) \wedge ... \wedge (B + i_0 < i_{d-1}) \wedge (i_{d-1} \leq B + 2 * i_0)\}.$$

Existential variables are those not declared as tuple variables or symbolic variables.

Uninterpreted function symbols, $f(p_1, p_2, ..., p_q)$, are functions whose value is unknown at compile time. As in [40], we assume that if $\vec{p} = \vec{x}$ then $f(\vec{p}) = f(\vec{x})$. We also allow the actual parameters p_k passed to any uninterpreted function symbol to be affine expressions of the tuple variables, symbolic constants, free variables, or uninterpreted function symbols, whereas in [40] uninterpreted function symbols are not allowed as parameters to other uninterpreted function symbols. In addition, in this prototype we require that the input domain and the output range for each uninterpreted function each be specified as a union of polyhedra that are not dependent on uninterpreted function symbols[1].

Abstract relations specify a set of integer tuple relation pairs with the same kinds of constraints allowed for abstract sets. For example, the following relation maps all three-dimensional tuples to a one-dimensional tuple where the value is their third element in the original tuple:

$$\{[i_0, i_1, i_2] \rightarrow [i_2]\}.$$

There are no constraints on the above relation so it is a set of infinite size with integer tuple pairs such as $\{[0, 0, 0] \rightarrow [0]\}$, $\{[0, 0, 1] \rightarrow [1]\}$, $\{[42, 7, 99] \rightarrow [99]\}$, etc. An abstract relation has an input tuple arity and an output tuple arity. As a notational convenience we subscript the names of abstract relations to indicate which sets are the domain and range of the relation. For example, the relation $A_{I \rightarrow X}$ has the abstract set I as its domain and abstract set X as its range.

Operations performed on abstract sets and relations include taking the inverse of a relation, applying a relation to a set, composing two relations, and taking the union or intersection of two relations or two sets. In [41], we provide more details about the implementation of these operations.

2.2. Specifying the Computation

Computations consist of symbolic constants, data and index arrays, statements, scheduling functions, access functions, and data dependences. This section describes each of these computation components in detail.

2.2.1. Symbolics, or Parameter Variables

Symbolic constants represent a constant value that is unchanging for the duration of the computation, but is not known at compile time. Examples of symbolics in Figure 1 are N_s, N_v, and N_e.

2.2.2. Data and Index Arrays

The SPF categorizes each array as either a data array or an index array. A *data array* typically contains the data being read and written within the computation and cannot be used to index into another array. An *index array* is an integer array that is used to index into data arrays or other index arrays.

[1] Our current implementation is restricted to the input domains being specified as a union of rectilinear domains and the output parameter being one-dimensional.

```
    for (s=0; s < N_s; s++) {
      for (i=0; i < N_v; i++) {
S1:       x[i] += ... fx[i] ... vx[i] ... ;
      }
      for (e=0; e < N_e; e++) {
S2:       fx[left[e]] += ... x[left[e]] ... x[right[e]] ... ;
S3:       fx[right[e]] += ... x[left[e]] ... x[right[e]] ... ;
      }
      for (k=0; k < N_v; k++) {
S4:       vx[k] += ... fx[k] ... ;
      }
    }
```

Figure 1: Simplified `moldyn` example.

Each data array has an associated *data space* represented with an abstract set with the same dimensionality as the array. The data space bounds can be affine functions of constants and symbolic constants. The original data space for the `x` array in Figure 1 is

$$x_0 = \{[m] \mid 0 \leq m < N_v\}.$$

The subscript "0" indicates that x_0 is the data space for data array `x` in the original, untransformed program. Note that the data space is the index domain of the data array.

Each index array is represented with an uninterpreted function symbol of the same name. As an uninterpreted function symbol in SPF, the domain of the index array, or its *index space*, must be specified along with the range of values that can be in the index array. For the index array `left` in Figure 1, its input domain is $\{[e] \mid (0 \leq e < N_e)\}$, and its output range is $\{[m] \mid (0 \leq m < N_v)\}$.

2.2.3. Statements

Computation occurs when statements access data and index arrays and apply various operations to them. Each iteration of a statement within a loop nest is represented as an integer tuple, $\vec{p} = [p_1, ..., p_n]$, where p_q is the value of the iteration variable for the qth loop in the loop nest. Thus, a statement's *original iteration space* is a polyhedral set of integer tuples with constraints indicating the affine loop bounds,

$$\{[p_1, ..., p_n] \mid lb_1 \leq p_1 \leq ub_1 \wedge \cdots \wedge lb_n \leq p_n \leq ub_n\}.$$

For statement S2 in Figure 1, the original iteration space is

$$I_{S2} = \{[s, e] \mid 0 \leq s < N_s \wedge 0 \leq e < N_e\}.$$

2.2.4. Scheduling Functions

In the SPF, a *scheduling function* maps each iteration of a statement into a shared iteration space. The schedule is then a lexicographical traversal of the points in the shared iteration space. Scheduling statements into imperfectly nested loops in this fashion was also used by Ahmed et al. [42], Kelly-Pugh [8], and is implemented as scattering functions in CLooG [43]. The statements in the simplified `moldyn` example in Figure 1 are mapped to a five-dimensional space (i.e., two dimensions for the loops and the other dimensions to denote loop and statement placement). The following relation specifies the scheduling function for statement S2 in Figure 1:

$$S_{I_{0,S2} \to \Phi_{0,S2}} = \{[s, e] \to [0, s, 1, e, 0]\},$$

where $I_{0,S2}$ denotes the original iteration space for statement $S2$ and $\Phi_{0,S2}$ denotes the shared iteration space. Each loop nest level corresponds to a pair of dimensions, where the first dimension of the pair is the numerical order of the loop as a statement, and the second dimension is a value of the index variable. The last value in the tuple corresponds to the statement's position with respect to other statements at the same level. The above scheduling function can be interpreted as first statement located within the second loop nested within the first loop when the iterator values are s and e.

We refer to the union of all the statement images in the shared iteration space as the full iteration space. Iteration reordering transformations are specified in terms of the full iteration space. The full iteration space is computed by applying the scheduling functions to each statement and then taking the union of the resulting sets.

The full iteration space Φ_0 for the (untransformed) program in Figure 1 is the following set:

$$\begin{array}{rcll} \Phi_0 = & \Phi_{0,S1} \cup \Phi_{0,S2} \cup \Phi_{0,S3} \cup \Phi_{0,S4} & & \\ = & \{[0, s, 0, i, 0] \mid & & (0 \le s < N_s) \wedge (0 \le i < N_v)\} \\ & \cup \{[0, s, 1, e, 0] \mid & & (0 \le s < N_s) \wedge (0 \le e < N_e)\} \\ & \cup \{[0, s, 1, e, 1] \mid & & (0 \le s < N_s) \wedge (0 \le e < N_e)\} \\ & \cup \{[0, s, 2, k, 0] \mid & & (0 \le s < N_s) \wedge (0 \le k < N_v)\}. \end{array}$$

For instance, using this representation, the $[s, k]$-th iteration of `S4` is denoted $[0, s, 2, k, 0]$ since `S4` is in the third statement (loop `k`) of the outer loop, and its the first statement within the `k` loop.

2.2.5. Access Functions

Given a specification of the original iteration space for each statement and its scheduling function, the next step is to specify how each statement accesses the data arrays. We define an *access function* as a function between the original iteration space for a statement and the storage location being accessed in data space a for a single memory access. We define an *access relation* $A_{I \to a}$ from sets of iterations to sets of storage locations into data space a, so that for each iteration $\vec{p} \in I$, $A_{I \to a}(\vec{p})$ is the set of locations that are referenced by iteration tuple $\vec{p}$. Notice that the subscript "$I \to a$" gives the domain and range of the mapping.

In the SPF, we use uninterpreted function symbols to abstractly represent the access relations that involve indirect array addressing through index arrays. The Figure 1 example has the following access relation for statement S2:

$$A_{I_{0,S2}\to x_0} \;=\; \{[s,e] \to [p] \mid p = left(e)\} \,\cup\, \{[s,e] \to [q] \mid q = right(e)\}.$$

The relation $A_{I_{0,S2}\to x_0}$ is the result of the two separate access functions (i.e., one for `x[left[e]]` and another for `x[right[e]]`) for S2 being unioned together into one relation for the whole statement.

Note that the relation $A_{I_{0,S2}\to x_0}$ is expressed in terms of the original iteration space for S2. Applying transformations to this access function requires that it be expressed in terms of the shared iteration space, $\Phi_{0,S2}$. The desired relation is therefore, $A_{\Phi_{0,S2}\to x_0}$.

$$\begin{aligned}
A_{\Phi_{0,S2}\to x_0} &= A_{I_{0,S2}\to x_0} \circ S^{-1}_{I_{0,S2}\to \Phi_{0,S2}} \\
&= A_{I_{0,S2}\to x_0} \circ S_{\Phi_{0,S2}\to I_{0,S2}} \\
&= \{[0,s,1,e,0] \to [p] \mid p = left(e)\} \,\cup\, \{[0,s,1,e,0] \to [q] \mid q = right(e)\}.
\end{aligned}$$

2.2.6. Data Dependences

The final step in specifying the computation is to specify the data dependences between iterations of statements in the original unified iteration space. The *dependence relation* $D_{\Phi\to\Phi} = \{\vec{p} \to \vec{q} \mid \text{ constraints } \}$ contains all pairs of iteration points in the full iteration space $\vec{p}, \vec{q} \in \Phi$ such that iteration $\vec{p}$ must execute before $\vec{q}$ due to a data dependence. It is also convenient to refer to subsets of $D_{\Phi\to\Phi}$ in terms of dependences between particular statements. We refer to subsets of $D_{\Phi\to\Phi}$ with the notation $d_{Sv,Sw}$, where v and w are statement numbers. For example, the dependences between statements `S1` ($[0,s,0,i,0]$) and `S2` ($[0,s,1,e,0]$) due to the `x` and `fx` arrays can be specified with the following dependence relation:

$$\begin{aligned}
d_{S1,S2} \;=\; &\{[0,s,0,i,0] \to [0,s',1,e,0] \mid (s \le s') \wedge i = left(e)\} \\
&\cup\{[0,s,0,i,0] \to [0,s',1,e,0] \mid (s \le s') \wedge i = right(e)\}.
\end{aligned}$$

2.3. Specifying RTRTs

The last section described how to express computations in the Sparse Polyhedral Framework (SPF) and this section describes how to express run-time reordering transformations (RTRTs) that can be applied to the computations. At compile time, the SPF enables the specification of RTRTs and the automatic determination of the effect an RTRT has on the scheduling function, access function, and data dependence specifications. The data and iteration reorderings that do not become explicit until runtime are expressed with the help of uninterpreted function symbols. At run-time the generated inspectors traverse and construct explicit relations to determine the current state of access functions, scheduling functions, and data dependences and to create reorderings and tilings, which are also stored as explicit relations. One of the key ideas in the SPF is that the effect of run-time reordering transformations can be expressed at compile time through formal manipulations of the computation specification (i.e., statement schedules, access functions, and data dependences), thus enabling the compile-time specification of a sequence of RTRTs.

2.3.1. Data Reorderings

Formally, a *data reordering transformation* is expressed at compile time with a data reordering specification $R_{a\to a'}$, where the data that was originally stored in some location m will be relocated to $R_{a\to a'}(m)$. The compile-time result of reordering an array a is that all access functions with the a data space as their range are modified to target the reordered data space a',

$$A_{\Phi\to a'} = \{\vec{p} \to R_{a\to a'}(m) \mid m \in A_{\Phi\to a}(\vec{p}) \wedge \vec{p} \in \Phi\}.$$

The above equation for $A_{\Phi\to a'}$ is equivalent to composing the data reordering relation $R_{a\to a'}$ with the access function $A_{\Phi\to a}$,

$$A_{\Phi\to a'} = R_{a\to a'} \circ A_{\Phi\to a}.$$

For example, assume that we apply a data permutation reordering to the data arrays `x` in Figure 1. The data reordering specification for data space `x` can be specified as follows:

$$R_{x_0\to x_1} = \{[p] \to [q] \mid q = \sigma(p)\},$$

where σ is an uninterpreted function symbol that denotes the data permutation reordering to be generated at runtime. At runtime, $R_{x_0\to x_1}$ can be realized with an explicit relation, which is a generalization of a one-dimensional index array.

The key idea in the SPF is that we can express at compile time how RTRTs will affect statement scheduling functions, access functions, and data dependences and therefore statically plan a series of such transformations and generate the code for an inspector and executor that implement the composition of a series of RTRTs. A data permutation reordering only affects access functions whose range is the reordered data space. Scheduling functions and data dependences are not affected because they relate iterations to time and iterations to iterations respectively. For the Figure 1 example, the $R_{x_0\to x_1}$ data permutation causes the incorporation of the σ uninterpreted function symbol into any access functions targeting the data array `x`. For example, the access relation for statement S2,

$$\begin{aligned} A_{\Phi_{0,S2}\to x_0} \quad &= \quad \{[0,s,1,e,0] \to [q] \mid q = left(e)\} \\ & \quad\quad \cup \{[0,s,1,e,0] \to [q] \mid q = right(e)\}, \end{aligned}$$

will become an access relation between the original full iteration space and the new `x` data space, x_1,

$$\begin{aligned} A_{\Phi_{0,S2}\to x_1} \quad &= \quad R_{x_0\to x_1} \circ A_{\Phi_{0,S2}\to x_0} \\ &= \quad \{[0,s,1,e,0] \to [q] \mid q = \sigma(left(e))\} \\ & \quad\quad \cup \{[0,s,1,e,0] \to [q] \mid q = \sigma(right(e))\}. \end{aligned}$$

Figure 2 shows how the executor code will change after applying the data reordering $R_{x_0\to x_1}$ to the `x` and `fx` data arrays (i.e., $R_{fx_0\to fx_1} = R_{x_0\to x_1}$).

```
    for (s=0; s < $N_s$; s++) {
      for (i=0; i < $N_v$; i++) {
S1:       x[$\sigma$[i]] += ... fx[$\sigma$[i]] ... vx[i] ... ;
      }
      for (e=0; e < $N_e$; e++) {
S2:       fx[$\sigma$[left[e]]] += ... x[$\sigma$[left[e]]] ... x[$\sigma$[right[e]]] ... ;
S3:       fx[$\sigma$[right[e]]] += ... x[$\sigma$[left[e]]] ... x[$\sigma$[right[e]]] ... ;
      }
      for (k=0; k < $N_v$; k++) {
S4:       vx[k] += ... fx[$\sigma$[k]] ... ;
      }
    }
```

Figure 2: Simplified `moldyn` example after reordering data arrays `x` and `fx` with σ.

2.3.2. Iteration Reorderings

An *iteration-reordering* transformation is expressed with a mapping $T_{\Phi\to\Phi'}$ that assigns each iteration $\vec{p}$ in iteration space Φ to iteration $T_{\Phi\to\Phi'}(\vec{p})$ in a new iteration space Φ'. The new execution order is given by the lexicographic order of the iterations in Φ'.

In the Figure 2 example, the σ data permutation of the `x` and `fx` arrays introduces indirect accesses to those arrays in the `i` and `k` loops. A transformation we call *iteration alignment* is an iteration permutation reordering that will cause the `i` and `k` loops to access the `x` and `fx` arrays sequentially in this example. The σ data permutation also introduced an additional level of indirection in the e loop, but we will remove that with a transformation called *pointer update* [24], which composes nested index arrays into a single index array.

The iteration alignment transformation is mathematically specified as a function on the full iteration space to a new full iteration space as seen here:

$$\begin{aligned} T_{\Phi_0\to\Phi_1} \quad = \quad & \{[0,s,0,i_0,0] \to [0,s,0,i_1,0] \mid i_1 = \sigma(i_0)\} \\ & \cup \{[0,s,1,e,q] \to [0,s,1,e,q] \mid 0 \le q \le 1\} \\ & \cup \{[0,s,2,k_0,0] \to [0,s,2,k_1,0] \mid k_1 = \sigma(k_0)\}. \end{aligned}$$

Notice that the transformation permutes the `i` and `k` loops, but does not affect the `e` loop. Also notice that this RTRT does not require a new explicit relation to be created at runtime, because it is using the reordering function σ that will be generated by the initial data permutation reordering transformation.

In general, an iteration reordering affects the scheduling function, access functions, and data dependences representing a computation. The scheduling function for a statement SX in the transformed iteration space Φ' is

$$S_{I_{SX}\to\Phi'_{SX}} = \{\vec{p} \to T_{\Phi\to\Phi'}(\vec{q})\}$$

or

$$S_{I_{SX}\to\Phi'_{SX}} = T_{\Phi\to\Phi'} \circ S_{I_{SX}\to\Phi_{SX}},$$

where $\{\vec{p} \to \vec{q}\} \in \Phi_{SX}$.

The dependences of the transformed iteration space are

$$D_{\Phi'\to\Phi'} = \{T_{\Phi\to\Phi'}(\vec{p}) \to T_{\Phi\to\Phi'}(\vec{q}) \mid \vec{p} \to \vec{q} \in D_{\Phi\to\Phi}\}$$

or

$$D_{\Phi'\to\Phi'} = T_{\Phi\to\Phi'} \circ (D_{\Phi\to\Phi} \circ T^{-1}_{\Phi\to\Phi'})$$

and the new access function $A_{\Phi'\to a}$ for each data space a is

$$A_{\Phi'\to a} = \{T_{\Phi\to\Phi'}(\vec{p}) \to A_{\Phi\to a}(\vec{p}) \mid \vec{p} \in \Phi\}$$

or

$$A_{\Phi'\to a} = A_{\Phi\to a} \circ T^{-1}_{\Phi\to\Phi'}.$$

Given the transformed access functions, scheduling functions, and dependences, we can specify further run-time reordering transformations (RTRTs).

In the Figure 2 example, the iteration alignment iteration permutation reordering $T_{\Phi_0\to\Phi_1}$ performs a loop permutation of the `i` and `k` loops. The effect of $T_{\Phi_0\to\Phi_1}$ on the scheduling function for statement S1

$$S_{I_{0,S1}\to\Phi_{0,S1}} = \{[s,i] \to [0,s,0,i,0]\}$$

is the following:

$$\begin{aligned} S_{I_{0,S1}\to\Phi_{1,S1}} &= T_{\Phi_0\to\Phi_1} \circ S_{I_{0,S1}\to\Phi_{0,S1}} \\ &= \{[s,i] \to [0,s,0,i_1,0] \mid i_1 = \sigma(i)\}. \end{aligned}$$

The transformed full iteration space will use i_1 as the iterator for the first loop nested within the `s` loop. There will be the constraint that $i_1 = \sigma(i)$, where i is an existential variable. Since σ is a permutation, the code generation process does not have to place a guard for the constraint $i_1 = \sigma(i)$ around S1.

The access function for statement S1 accessing array `x`, $A_{\Phi_{0,S1}\to x_1}\{[0,s,0,i,0] \to [q] \mid q = \sigma(i)\}$, becomes

$$\begin{aligned} A_{\Phi_{1,S1}\to x_1} &= \{[0,s,0,i,0] \to [q] \mid q = \sigma(i)\} \circ T^{-1}_{\Phi_0\to\Phi_1} \\ &= \{[0,s,0,i,0] \to [q] \mid q = \sigma(i)\} \circ \{[0,s,0,i_1,0] \to [0,s,0,i_0,0] \mid i_1 = \sigma(i_0)\} \\ &= \{[0,s,0,i_1,0] \to [q] \mid i_0 = i \wedge i_1 = \sigma(i_0) \wedge q = \sigma(i)\} \\ &= \{[0,s,0,i_1,0] \to [q] \mid i_1 = \sigma(i_0) \wedge q = \sigma(i_0)\} \\ &= \{[0,s,0,i_1,0] \to [q] \mid i_1 = q\}. \end{aligned}$$

Above we use the fact that σ is a permutation and therefore bijective to rewrite $q = \sigma(i_0)$ as $i_0 = \sigma^{-1}(q)$ and find that $i_1 = \sigma(\sigma^{-1}(q)) = q$.

The data dependences between statements 1 and 2,

$$\begin{aligned} d_{S1,S2} &= \{[0,s,0,i,0] \to [0,s',1,e,0] \mid (s \le s') \wedge i = left(e)\} \\ &\cup \{[0,s,0,i,0] \to [0,s',1,e,0] \mid (s \le s') \wedge i = right(e)\}. \end{aligned}$$

become

$$\begin{aligned} d_{S1,S2} &= \{[0,s,0,i,0] \to [0,s',1,e,0] \mid (s \le s') \wedge i = \sigma(left(e))\} \\ &\cup \{[0,s,0,i,0] \to [0,s',1,e,0] \mid (s \le s') \wedge i = \sigma(right(e))\}. \end{aligned}$$

Figure 3 shows the executor for the example code after iteration alignment.

```
    for (s=0; s < N_s; s++) {
      for (i_1=0; i_1 < N_v; i_1++) {
S1:       x[i_1] += ... fx[i_1] ... vx[σ^-1[i_1]] ... ;
      }
      for (j=0; j < N_e; j++) {
S2:       fx[σ[left[j]]] += ... x[σ[left[j]]] ... x[σ[right[j]]] ... ;
S3:       fx[σ[right[j]]] += ... x[σ[left[j]]] ... x[σ[right[j]]] ... ;
      }
      for (k_1=0; k_1 < N_v; k_1++) {
S4:       vx[σ^-1[k_1]] += ... fx[k_1] ... ;
      }
    }
```

Figure 3: Simplified `moldyn` example after aligning the loops `i` and `k` with the reordered data arrays `x` and `fx`.

2.4. Composing a legal sequence of RTRTs

A run-time reordering transformation (RTRT) specified in the SPF is legal if all current data dependences are respected in the new schedule. The compiler prototype presented here does not check transformation legality. However, the legality of an RTRT can be determined manually by using SPF to derive legality constraints on the uninterpreted functions generated by inspectors and then proving that an inspector implementation does satisfy the necessary constraints [44]. These proofs show that an inspector satisfies certain constraints for any possible input. Efforts are underway to automate such proofs [45]. Checking that such constraints are satisfied at runtime is not practical because it could significantly add to inspector overhead. Here we describe when dependences are affected by transformations, and how a sequence of transformations must coordinate.

Any *permutation* data reordering is legal in the SPF. If the data array `x` is permuted with the permutation σ, then all access functions targeting `x` can be updated with an additional indirect access. For example, `x[ ia[i] + ja[i] ]` would become `x[ sigma[ia[i] + ja[i]] ]`. Dependences between the data order and index arrays that occur in sparse matrix data structures such as compressed sparse row (i.e. the non-zeros for each row should be adjacent in the data array) are not allowed in the current prototype compiler for SPF due to the restriction that all loop bounds are affine expressions of the surrounding loop iterators. This means that computations over sparse matrix data structures other than coordinate storage will need to be flattened with some form of loop restructuring [46].

For iteration-reordering transformations, the new execution order must respect all the dependences of the original. Thus for each $\{\vec{p} \to \vec{q}\} \in D_{I\to I}$, $T_{I\to I'}(\vec{p})$ must be lexicographically earlier than $T_{I\to I'}(\vec{q})$,

$$\forall \vec{p}, \vec{q} : (\,\vec{p} \to \vec{q}\,) \in D_{I\to I} \Rightarrow T_{I\to I'}(\vec{p}) \prec T_{I\to I'}(\vec{q}).$$

Lexicographical order on integer tuples can be defined as follows [47]:

$$[p_1, ..., p_n] \prec [q_1, ..., q_n] \Leftrightarrow$$
$$\exists m : \ (\forall i : \ 1 \le i < m \Rightarrow p_i = q_i) \wedge (p_m < q_m).$$

Since the dependences may involve uninterpreted function symbols, compile-time legality checking is not straightforward. It requires computing pre and post conditions that individual explicit relations or index arrays must satisfy for the transformation to be legal and then either checking those conditions at runtime or performing a compile-time pre and post condition analysis of the run-time library routines that generates the explicit relations or index arrays in question. We show how this can be done for a sparse tiling of the Gauss-Seidel computation in [48].

The compiler creates a composition of transformations, but currently each transformation is specified individually. Between transformations there are some simpler legality checks that can be leveraged to provide helpful error messages to users of the sparse polyhedral framework. These checks include determining if an iteration transformation has been properly specified for the full iteration space, checking that any run-time reorderings are providing input of the appropriate domain to uninterpreted function symbols, ensuring that uninterpreted function symbols are placed in equality constraints with expressions whose domain matches the function range, and verifying that an iteration transformation matches the dimensionality of the full iteration space. As an example of the last check, if the first transformation maps the iteration space into a 2D iteration space, then the second transformation on the iteration space must map a 2D iteration space to its target.

2.5. Example RTRT Compositions

Data and iteration reordering RTRTs can be applied in a sequence. Appropriate composition of the transformations with the statement schedule functions, access functions, and data dependences determines the effect of the transformations on the resulting executor code. Figure 4 summarizes the examples of the data permutation *consecutive packing* (*cpack*) and the iteration permutation *iteration alignment* described in Sections 2.3.1 and 2.3.2. The net effect of *cpack* followed by *iteration alignment* on the executor code can be seen in the code fragment that includes statement S1. The schedule function is implemented with appropriate loop nesting, and the access functions specify the index expressions for data array accesses.

Figures 5 and 6 summarize some subsequent RTRTs for the running example. Figure 5 shows how a data permutation transformation called *data alignment* removes the additional indirect reference through σ^{-1} that was introduced due to consecutive packing followed by iteration alignment. Figure 5 also shows the effect of an iteration permutation on the `e` loop. For this iteration permutation, the user indicates which loop should be permuted based on how that loop is accessing certain data arrays (e.g., `x` and `fx` in the example). One possible iteration permutation reordering algorithm is locality grouping [24]. The reordering algorithm selected is responsible for generating the δ permutation at runtime. In the executor, the statements all maintain the same scheduling function, because the transformation is an iteration permutation, which does not require changing the loop structure. In other words, the loop being permuted will still need the same bounds. The permutation of the iterations is reflected

Name	RTRT class Input Abstract Relations Transformation Specification Composed effect on executor for Figure 1
cpack	data permutation on `x` and `fx`
	$A_{I_e \to x_0} = \{[e] \to [q] \mid q = left(e) \wedge 0 \leq e < N_e\}$ $\cup \{[e] \to [q] \mid q = right(e) \wedge 0 \leq e < N_e\}$
	$R_{x_0 \to x_1} = \{[p] \to [q] \mid q = \sigma(p)\}$

	for (s=0; s < N_s; s++) {
	  for (i=0; i < N_v; i++) {
	S1: x[σ[i]] += ... fx[σ[i]] ... vx[i] ... ;
	  }
	  for (e=0; e < N_e; e++) {
	S2: fx[σ[left[e]]] += ... x[σ[left[e]]]
	        ... x[σ[right[e]]] ... ;
	  ...

iter align	iteration permutation on `i` and `k`
	since the σ function is already available, this transformation does not have a run-time component that needs input
	$T_{I_0 \to I_1} = \{[0, s, 0, i_0, 0] \to [0, s, 0, i_1, 0] \mid i_1 = \sigma(i_0)\}$ $\cup \{[0, s, 1, e, q] \to [0, s, 1, e, q]\}$ $\cup \{[0, s, 2, k_0, 0] \to [0, s, 2, k_1, 0] \mid k_1 = \sigma(k_0)\}.$

	for (s=0; s < N_s; s++) {
	  for (i_1=0; i_1 < N_v; i_1++) {
	S1: x[i_1] += ... fx[i_1] ... vx[σ^{-1}[i_1]] ... ;
	  ...

Figure 4: Summary of the data permutation and iteration permutation examples described in Sections 2.3.1 and 2.3.2. Includes the RTRT specification, the specification of the input for the run-time reordering algorithm, and the RTRT's effect on parts of the executor code.

in changes to the access relations and the data dependences (i.e., instead of e using $\delta^{-1}(e_2)$). Note that pointer update [24] is used to compose nested index arrays into a single index array.

Figure 6 summarizes an RTRT called sparse tiling. A *sparse tiling* is a transformation that maps a space of iteration points into a set of tiles. The new schedule for the iteration space is then to execute the iteration points by tile. Therefore, the transformed code includes a new loop that iterates over the tiles. One goal of a sparse tiling transformation is to group iterations such that iterations that reuse the same data are within the same tile and therefore

Name	**RTRT class**
	Input Abstract Relations
	Transformation Specification
	Effect on example computation in Figure 1
data align	data permutation on vx
	no input abstract relations
	$R_{vx_0 \to vx_1} = \{[p] \to [q] \mid q = \sigma(p)\}$

	for (s=0; s < N_s; s++) {
	  for (i_1=0; i_1 < N_v; i_1++) {
	S1: x[i_1] += ... fx[i_1] ... vx[i_1] ... ;

locality grouping	iteration permutation on the e loop based on accesses to x
	$A_{I_e \to x_1} = \{[e] \to [q] \mid q = \sigma(left(e))\} \cup \{[e] \to [q] \mid q = \sigma(right(e))\}$
	$T_{I_1 \to I_2} = \{[0,s,0,i,0] \to [0,s,0,i,0]\} \cup \{[0,s,1,e_1,q] \to [0,s,1,e_2,q] \mid e_2 = \delta(e_1)\} \cup \{[0,s,2,k,0] \to [0,s,2,k,0]\}.$

	for (s=0; s < N_s; s++) {
	  ...
	  for (e_2=0; e_2 < N_e; e_2++) {
	S2: fx[σ[left[δ^{-1}[e_2]]]] += ... x[σ[left[δ^{-1}[e_2]]]]
	      ... x[σ[right[δ^{-1}[e_2]]]] ... ;

Figure 5: Sequence of RTRTs applied to e loop after cpack and iteration alignment. Data alignment is applied to array vx and an iteration permutation is applied to the e loop.

the computation as a whole can experience improved temporal data locality. Another possible goal is to create task-level parallelism.

In Figure 6, we sparse tile across the i, e, and k loops. The sparse tiling algorithm partitions the iterations in one of those loops and then place iterations from the other loops into tiles so that when the tiles are executed in order, the dependences of the computation are satisfied. Note that the dependences between the i and e, and the e and k loops are input to the sparse tiling inspector that will execute at runtime. Full sparse tiling [28, 49] is one possible sparse tiling algorithm that places iterations into disjoint tiles. Unstructured cache

<table>
<tr><th rowspan="4">Name</th><th>RTRT class</th></tr>
<tr><th>Input Abstract Relations</th></tr>
<tr><th>Transformation Specification</th></tr>
<tr><th>Effect on example computation in Figure 1</th></tr>
<tr><td rowspan="4">sparse tiling</td><td>groups iterations across loops i, e, and k based on dependences between those loops</td></tr>
<tr><td>$$\begin{aligned} D_{I_2 \to I_2} \;=\; & \{[0,s,0,i,0] \to [0,s,1,e,q] \mid i = \sigma(left(\delta^{-1}(e)))\} \\ & \cup \{[0,s,0,i,0] \to [0,s,1,e,q] \mid i = \sigma(right(\delta^{-1}(e)))\} \\ & \cup \{[0,s,1,e,q] \to [0,s,2,k,0] \mid k = \sigma(left(\delta^{-1}(e)))\} \\ & \cup \{[0,s,1,e,q] \to [0,s,2,k,0] \mid k = \sigma(right(\delta^{-1}(e)))\} \end{aligned}$$</td></tr>
<tr><td>$$\begin{aligned} T_{I_2 \to I_3} \;=\; & \{[0,s,0,i,q] \to [0,s,0,t,0,i,q] \mid t = \theta(0,i)\} \\ & \cup \{[0,s,1,e,q] \to [0,s,0,t,1,e,q] \mid t = \theta(1,e)\} \\ & \cup \{[0,s,2,k,q] \to [0,s,0,t,2,k,q] \mid t = \theta(2,k)\} \end{aligned}$$</td></tr>
<tr><td><pre>
     for (s=0; s < N_s; s++) {
       for (t=0; t < N_t; t++) {
         for (i=0; i < N_v; i++) {
S1:   if (t == θ(0,i)) { x[i] = ... fx[i] ... vx[i] ... ; }
         }
         for (e=0; e < N_e; e++) {
S2:   if (t == θ(1,e)) { fx[σ[left[δ^{-1}[e]]]]
                          += ... x[σ[left[δ^{-1}[e]]]]
                             ... x[σ[right[δ^{-1}[e]]]] ... ; }
S3:   if (t == θ(1,e)) { fx[σ[right[δ^{-1}[e]]]]
                          += x[σ[left[δ^{-1}[e]]]]
                           ... x[σ[right[δ^{-1}[e]]]] ...; }
         }
         for (k=0; k < N_v; k++) {
S4:   if (t == θ(2,k)) { vx[k] += ... fx[k] ... ; }
         }
     }
</pre></td></tr>
</table>

Figure 6: Sparse tiling RTRT applied to i, e, and k loops after all data and iteration permutations.

blocking [50] is another approach. The communication-avoiding algorithms of Demmel et al. [51] also create sparse tiles, but those tiles overlap so as to enable parallel execution of the tiles and minimal communication between tiles.

A sparse tiling inspector creates an explicit function, which we call θ, that maps points in an iteration sub-space to tiles. Note that in the resulting code in Figure 6 the tiling function θ is used to guard statements in the i, e, and k loops. Guard encapsulation [52] removes the guards and makes the i, e, and k loops only execute the iterations specific to the current tile by using a sparse data structure similar to compressed sparse row (CSR).

2.6. Sparse Polyhedral Framework Summary

A transformation framework provides a formal way to represent all aspects of the transformation process. The Sparse Polyhedral Framework (SPF) represents computations with indirect memory accesses and run-time reordering transformations with integer tuple sets and relations with affine constraints and constraints involving uninterpreted function symbols. A composed transformation is a sequence of data and iteration transformation mappings. The reordering heuristics that the inspector will apply for each transformation use as input the transformed data dependences and access functions that result from all previous transformations. A composition of transformations is legal if the final data dependences can be shown to be lexicographically positive, and it is possible to check post-conditions on the reordering functions generated by inspectors. This section shows how the SPF could be used to represent a molecular dynamics computation and various RTRT transformations.

The SPF can be used to generate an inspector containing all of the run-time reordering algorithms being applied in the appropriate order and an executor that implements the transformed code and uses the reordering functions provided by the inspector. The next sections describe how we generate code for composed inspectors and their corresponding executors and enable the authoring of run-time reordering transformations (RTRTs).

3. Inspector/Executor Code Generation

Run-time reordering transformations are typically implemented with inspector/executor strategies. When a series of RTRTs are expressed within the Sparse Polyhedral Framework (SPF) as shown in Section 3, the code for *most* of the inspector and all of the executor can be automatically generated. This section presents intermediate representations for both the inspector and executor, a run-time library that provides support for the inspectors, and algorithms for generating inspector and executor code.

3.1. Intermediate Representations (IRs) for RTRTs, Inspectors, and Executors

As is typical with a transformation framework, SPF includes a mechanism for specifying the original computation and data spaces and transformations on those spaces. The Mapping IR is a data structure that implements the SPF and as such represents the executor, which is a (un)transformed version of the original computation. The Mapping IR also represents the run-time reordering transformations (RTRTs) as a sequence of data and/or iteration reordering relations. The Inspector Dependence Graph (IDG) represents various computations the inspector must perform to generate the necessary reordering functions and relations and reorder data.

3.1.1. The Mapping IR (MapIR)

The Mapping Intermediate Representation (MapIR) encodes the computation specification, which includes statements, symbolic constants, data and index arrays, access relations, and data dependences. Section 2.2 describes the computation specification components of the Sparse Polyhedral Framework (SPF) in detail. The MapIR implementation in our prototype Inspector/Executor Generator Python prototype (IEGen in Python) provides a Python interface for specifying integer tuple sets and relations for the various components of the computation specification. As an example, the index array `left` in the molecular dynamics example in Figure 1 can be specified in the IEGen Python prototype as follows:

```
spec.add_index_array(
    name='left',
    type='int * %s',
    input_bounds='{[q]: 0<=q && q<N_e}',
    output_bounds='{[q]: 0<=q && q<N_v}')
```

The RTRTs are represented in the MapIR as a sequence of iteration and data reordering relations. For iteration reorderings, the transformation is specified for the full iteration space. For data reorderings, information about which data spaces will be affected by the reordering is included. Figure 3.1.1 summarizes the sequence of transformations for the molecular dynamics example.

When transformations are applied to the computation, they modify the statement scheduling functions, access functions, and data dependences in the MapIR to indicate their compile-time effect on the computation. Section 2.3 formalizes the effect of data and iteration reordering transformations on scheduling functions, access relations, and data dependences. A transformation implemented in IEGen Python uses the mathematical framework provided by SPF to automate the application of run-time reordering transformations (RTRT).

3.1.2. Inspector Dependence Graph (IDG)

In addition to the application of a transformation modifying the computation specification in the MapIR, each transformation typically involves run-time reordering functionality that the inspector will perform. Therefore, the application of a sequence of transformations leads to a set of related inspector tasks. The Inspector Dependence Graph (IDG) represents these tasks, the data structures consumed and generated by the inspector, and the dependences between data and tasks within the inspector.

Figure 8 shows an example IDG, where the rectangular nodes represent data structures and the elliptical nodes represent tasks. An edge that starts at a data node and ends at a task node indicates that the task will be using that data. An edge that starts at a task node and ends at a data node indicates that the task will be generating that data.

The IDG in Figure 8 represents the inspector that along with the corresponding executor implements the consecutive packing and iteration alignment

Name	**Transformation Specification**
cpack data reordering	$R_{x_0 \to x_1} = R_{fx_0 \to fx_1} = \{[p] \to [q] \mid q = \sigma(p)\}$
iteration alignment	$T_{I_0 \to I_1} = \{[0, s, 0, i_0, 0] \to [0, s, 0, i_1, 0] \mid i_1 = \sigma(i_0)\}$ $\cup \{[0, s, 1, e, q] \to [0, s, 1, e, q]\}$ $\cup \{[0, s, 2, k_0, 0] \to [0, s, 2, k_1, 0] \mid k_1 = \sigma(k_0)\}$
data alignment	$R_{vx_0 \to vx_1} = \{[p] \to [q] \mid q = \sigma(p)\}$
locality grouping iteration reordering	$T_{I_1 \to I_2} = \{[0, s, 0, i, 0] \to [0, s, 0, i, 0]\}$ $\cup \{[0, s, 1, e_1, q] \to [0, s, 1, e_2, q] \mid e_2 = \delta(e_1)\}$ $\cup \{[0, s, 2, k, 0] \to [0, s, 2, k, 0]\}.$
sparse tiling	$T_{I_2 \to I_3} = \{[0, s, 0, i, q] \to [0, s, 0, t, 0, i, q] \mid t = \theta(0, i)\}$ $\cup \{[0, s, 1, e, q] \to [0, s, 0, t, 1, e, q] \mid t = \theta(1, e)\}$ $\cup \{[0, s, 2, k, q] \to [0, s, 0, t, 2, k, q] \mid t = \theta(2, k)\}$

Figure 7: Sequence of data and iteration reordering transformations that are applied in the running example in Figures 4, 5, and 6.

RTRTs summarized in Figure 4. Recall that in the molecular dynamics example, the interactions between atoms are encoded in the index arrays `left` and `right`, where `left[i]` and `right[i]` are the indices for interacting atoms. In Figure 4, the input to the consecutive packing data reordering heuristic is an *abstract relation* describing how the `e` loop is accessing the `x` and `fx` arrays associated with atoms. The IDG in Figure 8 shows that the `left` and `right` index arrays are used as input to an inspector task that will construct an *explicit relation*, which will then be passed to the consecutive data reordering algorithm to generate the σ explicit relation that represents the data permutation. After reordering σ has been generated, the `reorderArray` function applies the σ permutation to the `x` and `fx` data arrays. Note that the data arrays in the IDG include a version number to represent versions of the same array, where the array is undergoing in-place data reorderings.

In the example, the application of iteration alignment can be performed entirely at compile time as modifications to the access functions for the `x` and `fx` arrays in the `i` and `k` loops. Therefore, no tasks are added to the IDG for iteration alignment in this example, because the σ uninterpreted function has already been generated by the inspector, and the cancelation of σ by σ^{-1} in the access function for the array `vx` occurs at compile time.

In general, there are two main kinds of computation nodes within the IDG: explicit relation generation loops and function calls. *Explicit relation generation*

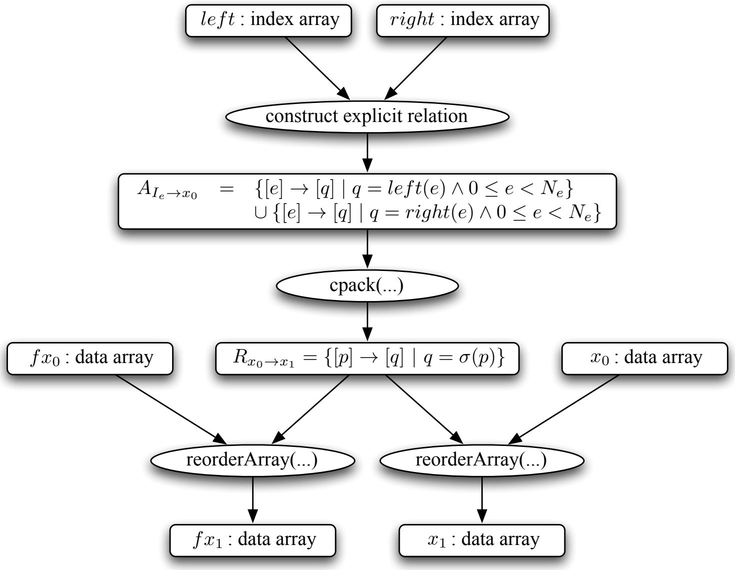
$left$: index array
$right$: index array
construct explicit relation
$A_{I_e \to x_0} = \{[e] \to [q] \mid q = left(e) \wedge 0 \leq e < N_e\} \cup \{[e] \to [q] \mid q = right(e) \wedge 0 \leq e < N_e\}$
cpack(...)
fx_0 : data array
$R_{x_0 \to x_1} = \{[p] \to [q] \mid q = \sigma(p)\}$
x_0 : data array
reorderArray(...)
reorderArray(...)
fx_1 : data array
x_1 : data array

Figure 8: The Inspector Dependence Graph (IDG) after the compile-time application of data permutation on the data arrays `x` and `fx` based on how `x` and `fx` are accessed in the `e` loop.

loops are loops that construct an explicit relation at runtime. These loops are automatically generated by our code generator prototype called IEGen Python. These loops iterate over the domain of the abstract relation (e.g. the domain of $A_{I_e \to x_0}$ is $\{[e] \mid 0 \leq e < N_e\}$) and compute all of the relations for insertion into the explicit relation data structure. *Function call nodes* within the IDG represent function calls to run-time library routines either written by the transformation writer to support a transformation (e.g. cpack) or general run-time support routines such as `reorderArray`.

3.2. Explicit Relation Run-Time Library

Run-time Reordering Transformations (RTRT) consist of compile-time and a run-time components. The compile-time components include an interface for the RTRT user to specify the abstract relation for the transformation and any transformation-specific parameters. Other compile-time components include routine(s) for modifying the MapIR and IDG to show the effects of the transformation. The run-time component of a transformation includes any routines that the inspector will call at runtime. In the example, the `reorderArray` utility routine takes a one-dimensional array with specified size and element size and a permutation and then reorders the array. Also, as can be seen in the example IDGs in Figures 8, 10, and 11, explicit instances of sets and relations are built in the inspector as inputs and outputs of the reordering algorithms

```
void ERG_cpack(ER_U1D* inputRelation, EF_1D* sigma) {
    // assigned[i] indicates whether the value i has been reordered
    bool *assigned;
    int N = EF_1D_in_domain_size(sigma);
    assigned = (int*)malloc(sizeof(int)*N);
    for (i=0; i<N; i++) assigned[i]=false;

    // Loop over the [in] -> [out] and reorder out values
    // based on a first-come-first-served policy.
    int count = 0;
    for (int in=ER_U1D_in_domain_lb(inputRelation);
           in<=ER_U1D_in_domain_ub(inputRelation);
           in++)
    {
        for (int out=ER_U1D_out_begin(inputRelation,in);
              out!=ER_U1D_out_end(inputRelation,in);
              out=ER_U1D_out_next(inputRelation,in))
        {
            if (!taken[out]) {
                EF_1D_set(sigma, out, count);
                assigned[out]   = true;
                count++;
            }
        }
    }

    // Reorder any leftover values in the output domain.
    for (int i=0; i<N; i++) {
      if (!assigned[i]) EF_1D_set(sigma, count++);
    }
}
```

Figure 9: Consecutive packing inspector that uses specialized implementations of the explicit relation data structure for performance reasons, but not specific to any single input code.

such as consecutive packing (cpack), locality grouping (locgroup), and sparse tiling (fullSparseTile). These run-time components of the run-time reordering transformations are performed by the generated inspector code with support from a run-time library with routines that manipulate the explicit relation data structure and that implement reordering algorithms that operate over instances of the explicit relation data structure.

The explicit relation *abstract data type* represents any m-to-n-dimensional relation and is the core concept in the IEGen run-time library. By using the explicit relation concept, the run-time library routines do not need to be specific to data structures within each application being transformed. The IEGen Python prototype generates the parts of the inspector that are specific to an individual application such as the names of index and data arrays. However, since a fully general explicit relation data structure is not efficient enough to compete with inspectors written for specific index array usage, our prototype run-time library contains the following specializations:

- explicit relations that are functions and have 1D to 1D arity (`EF_1D`),
- explicit relations that have 1D to 1D arity and can be represented as a union of 1D to 1D explicit functions (`ER_U1D`),

- explicit relations with 1D to 1D arity and 2D to 1D arity where the relations are not inserted in the order of the input tuples (`ER_1Dto1D` and `ER_2Dto1D`),
- explicit relations that are functions and have 2D to 1D arity (`EF_2D`), and
- explicit relations that represent 1D to 1D arity dependences between loops (`ExplicitDependence`).

Future work includes automating the process of specializing the explicit relation implementations.

3.3. Code Generation for Inspectors

The inspector code generation algorithm consists of three topological visits of the IDG where the computation nodes (ellipses) in the IDG trigger the generation of explicit relation construction loops or function calls, and the data nodes (rectangles) trigger the generation of the appropriate parameter list, variable declarations, and deallocation code at the end of the inspector. The first pass over the IDG determines which data nodes have no incoming or outgoing edges and therefore will become parameters to the inspector function. The one exception for this selection of parameters is that data arrays are represented as multiple versions in the IDG and only one instance of the data array exists at any one time during the execution so only one parameter per data array is necessary. During the second pass over the IDG, the inspector code generator produces an explicit relation declaration and initialization for each of the explicit relation and index data nodes. Specialized explicit relation implementations are selected based on the characteristics of the corresponding abstract relation. As a final step, we generate the main body of the inspector and cleanup code by performing a topological visit to all of the computational nodes and keeping track of which IDG data nodes are only used within the IDG and therefore need to be deallocated at the end of the inspector.

3.4. Code Generation for Executors

The two main steps of executor code generation are: statement generation and loop structure generation. In the IEGen Python prototype, each statement is represented as a string with holes for access functions. Additionally, each statement has an iteration space and a scheduling function that maps the statement iteration space to a full iteration space that includes all statements. To generate each statement, we plug the access function holes with the transformed access functions. The statement is defined as a C macro with the iterators of the loop as input parameters to the macro. We use CLooG [43] to generate loops that scan all of the iteration points in the part of the iteration space that is constrained by affine constraints.

As CLooG is not able to generate code to iterate over sparse sets, we have a final step that adds code to do this within the IEGen Python prototype. Any constraints involving uninterpreted function symbols equalities in the executor set representation will be placed in the statement macro as a wrapper around the new version of the statement. Figure 6 shows an example of updated data array references and uninterpreted function constraints resulting in `if`-statement guards within the innermost loops of the computation. A portion of that example is repeated here for illustrative purposes:

```
    for (s=0; s < $N_s$; s++) {
      for (t=0; t < $N_t$; t++) {
        for (i=0; i < $N_v$; i++) {
S1:       if (t == $\theta$(0,i)) { x[i] = ... fx[i] ... vx[i] ... ; }
        }
        ...
    }
```

Introducing guards into the innermost loops is a performance problem because guards cause a conditional branch within the innermost loops and because they result in a significant amount of loop overhead since many more iterations are visited than actually executed. For the molecular dynamics example, the number of iterations after straight-forward sparse tiling code generation is the number of tiles times the number of original iterations in each of the `i`, `e`, and `k` loops, which is significantly greater than the number of original iterations. Guard encapsulation [52] solves this performance overhead problem.

Past work has included run-time reordering transformations (RTRTs) for which a compiler can automatically analyze and generate the inspectors [33, 24, 25, 30] for specific run-time reordering transformations. Through the manipulation of the SPF abstract sets and relations at compile time and the use of explicit relation data structures at run time, we are able to generate inspectors and executors for more general compositions of RTRTs.

4. Authoring RTRTs

The SPF can be thought of as the assembly-language level for specifying Run-Time Reordering Transformations (RTRTs). Much like in the CHill project [53], we suggest that specific RTRTs should be made available to performance programmers as higher-level concepts such as "consecutive packing based on the memory references in loop `e`" and "sparse tiling of the three loops using the second loop as the seed partition". Therefore the IEGen Python prototype code generator provides implementations of the abstract relations and sets in addition to the explicit relation implementation in the run-time library, so that transformation writers can provide a higher-level interface to users. This section describes how a transformation writer might implement the data reordering consecutive packing and the iteration reordering full sparse tiling.

Our experiences with the IEGen Python prototype is that authoring and using the RTRT transformations require an expert user. More work is needed to ease the use of the IEGen transformation tool. Possible improvements include automating the data dependence analysis based on previous research in value-based dependence analysis [54, 55, 56, 57] and dependence analysis in irregular applications [58], and computing the SPF transformation specification based on high-level information such as which data arrays should be reordered and which loops should be sparse tiled.

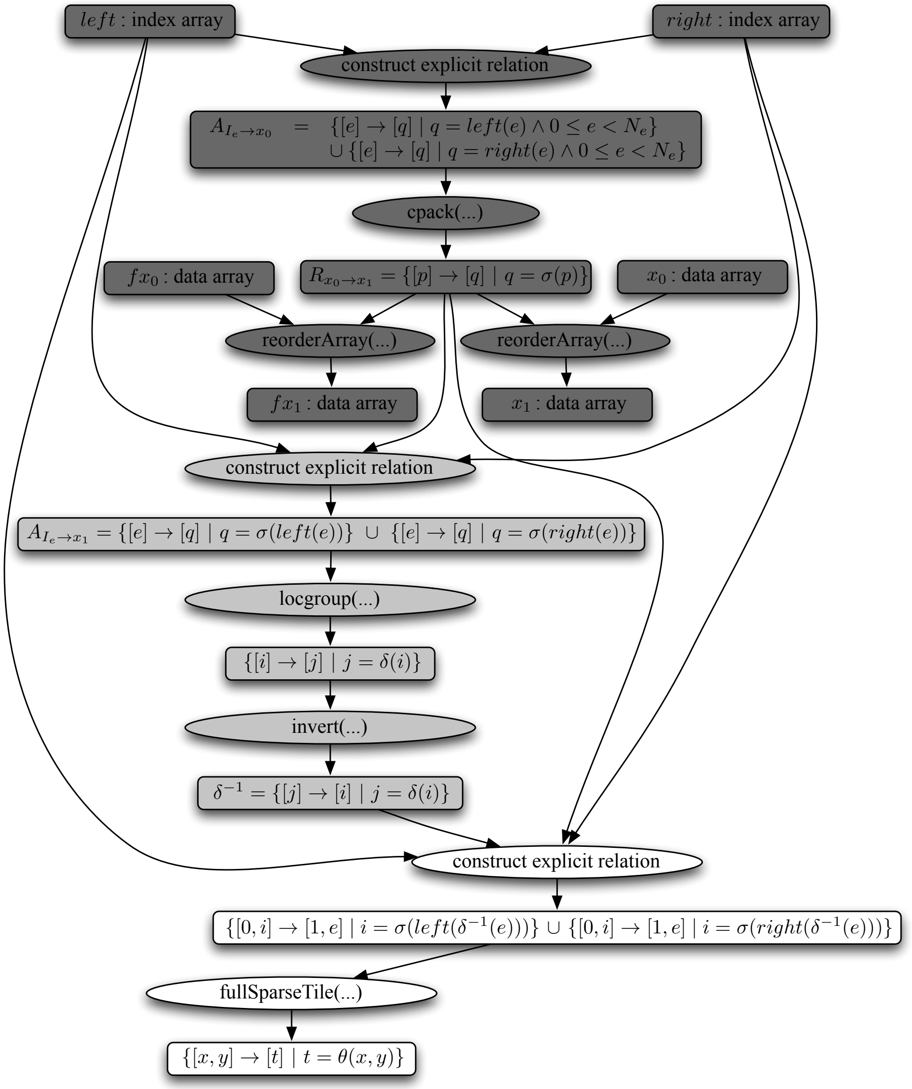
$left$: index array
$right$: index array
construct explicit relation
$A_{I_e \to x_0} = \{[e] \to [q] \mid q = left(e) \wedge 0 \le e < N_e\} \cup \{[e] \to [q] \mid q = right(e) \wedge 0 \le e < N_e\}$
cpack(...)
fx_0 : data array
$R_{x_0 \to x_1} = \{[p] \to [q] \mid q = \sigma(p)\}$
x_0 : data array
reorderArray(...)
reorderArray(...)
fx_1 : data array
x_1 : data array
construct explicit relation
$A_{I_e \to x_1} = \{[e] \to [q] \mid q = \sigma(left(e))\} \cup \{[e] \to [q] \mid q = \sigma(right(e))\}$
locgroup(...)
$\{[i] \to [j] \mid j = \delta(i)\}$
invert(...)
$\delta^{-1} = \{[j] \to [i] \mid j = \delta(i)\}$
construct explicit relation
$\{[0,i] \to [1,e] \mid i = \sigma(left(\delta^{-1}(e)))\} \cup \{[0,i] \to [1,e] \mid i = \sigma(right(\delta^{-1}(e)))\}$
fullSparseTile(...)
$\{[x,y] \to [t] \mid t = \theta(x,y)\}$

Figure 10: The inspector dependence graph after the compile-time application of sparse tiling on the `i` and `e` loops based on the dependences between the `i` and `e` loops.

4.1. Data Reordering Example

A transformation writer is responsible for (1) providing the user a way to specify transformation parameters, (2) providing an implementation of the transformation that modifies the inspector and executor intermediate representations the IDG and MapIR to reflect the effect of the transformation, and (3) providing any functions needed for the run-time generation of reorderings. For an example instance of each of these transformation components, we describe applying data reordering to the molecular dynamics example in Figure 1.

In Figure 1, the `e` loop is accessing the `x` and `fx` arrays in an irregular fashion. Therefore a data permutation reordering of the `x` and `fx` arrays could improve spatial locality and consequently the performance in the loop. The parameters for a data reordering permutation transformation include an indication of which data arrays should be permuted (i.e. `x` and `fx`) and which access functions should be inspected to determine a heuristic permutation (i.e. the access relation between the `e` loop and the `x` and `fx` arrays).

The transformation writer implements the transformation by enabling the user to specify the needed parameters and then using those parameters to modify the inspector and executor intermediate representations, the IDG and the MapIR. In the molecular dynamics example, the access functions between the `e` loop for data arrays `x` and `fx` are used as input to the data reordering for the creation of σ (see Figure 8). A user of the data permutation transformation provides parameters indicating the run-time reordering algorithm to use (e.g. cpack), the access functions to use as input to the reordering algorithm (e.g. $A_{I_e \to x_0}$ in Figure 8), and the data arrays that should be reordered based on the generated data permutation (e.g. `x` and `fx`).

Given the transformation parameters, the transformation compile-time component is responsible for modifying the MapIR and IDG representations to record the compile-time effect of the RTRT. For example, the data permutation transformation creates the initial IDG in Figure 8. There are utility functions available in the IEGen Python prototype that help ease the task of constructing subgraphs within the IDG and connecting nodes with edges. A transformation modifies the MapIR by leveraging the Sparse Polyhedral Framework, which indicates the effect of data and iteration reordering transformations on access functions, scheduling functions, and data dependences. For the example, the cpack data permutation transformation modifies the access functions as shown indirectly in Figure 4 (i.e., `x[i]` becomes `x[`σ`[i]]`).

4.2. Sparse Tiling Reordering Example

For a more complex example, consider the sparse tiling transformation whose modifications to the MapIR are shown indirectly in Figure 6 and whose modifications to the IDG are shown in the white nodes of Figure 10 (e.g., `x[i] = ... fx[i] ... vx[i] ... ;` becomes `if (t ==` θ`(0,i)) x[i] = ... fx[i] ... vx[i] ... ;`). The full sparse tiling transformation applied to the moldyn example schedules some iterations of each of the `i`, `e`, and `k` loops to be executed atomically before moving on to another tile with the goal of improving temporal locality and possibly exposing task graph parallelism.

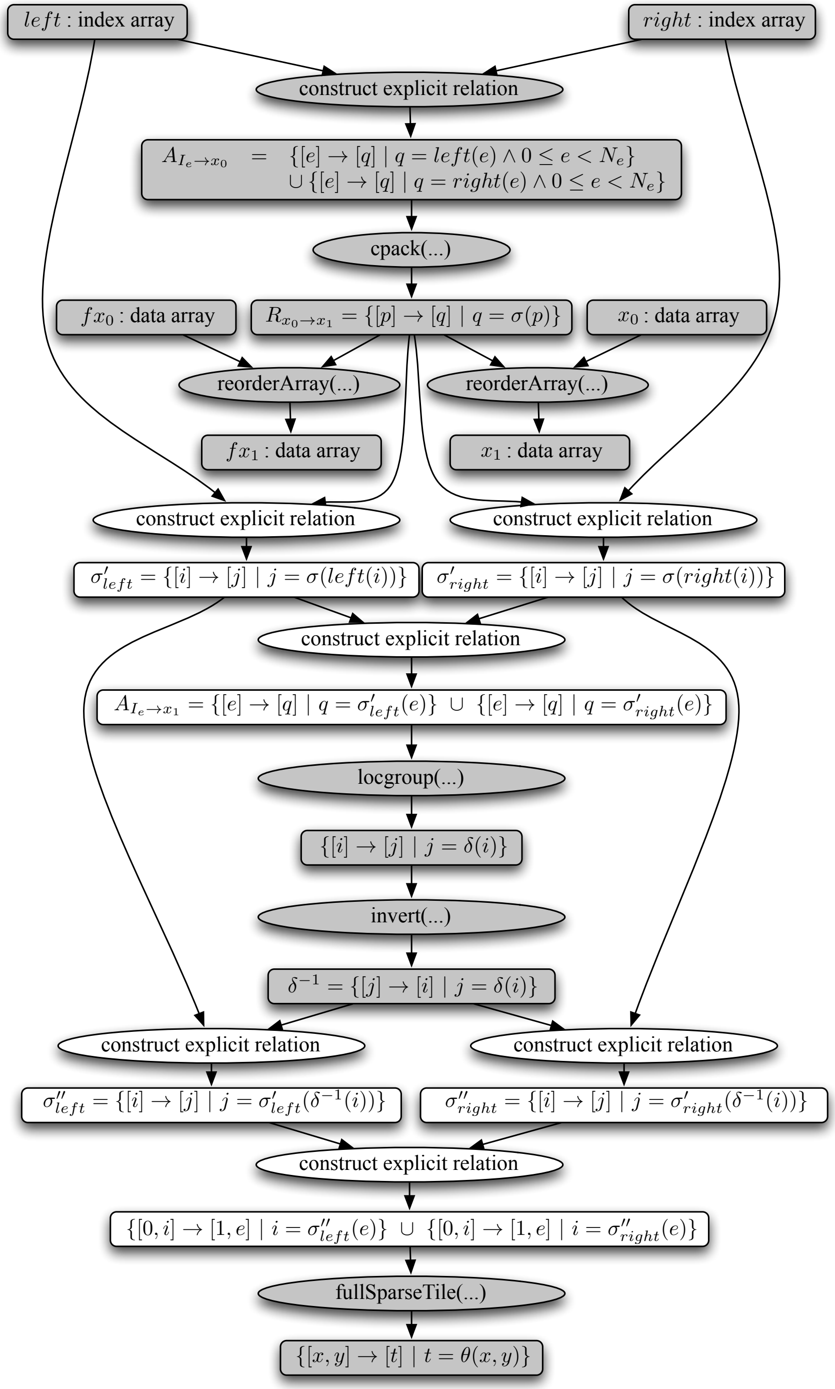
$left$: index array
$right$: index array
construct explicit relation
$A_{I_e \to x_0} = \{[e] \to [q] \mid q = left(e) \wedge 0 \leq e < N_e\}$
$\cup \{[e] \to [q] \mid q = right(e) \wedge 0 \leq e < N_e\}$
cpack(...)
fx_0 : data array
$R_{x_0 \to x_1} = \{[p] \to [q] \mid q = \sigma(p)\}$
x_0 : data array
reorderArray(...)
reorderArray(...)
fx_1 : data array
x_1 : data array
construct explicit relation
construct explicit relation
$\sigma'_{left} = \{[i] \to [j] \mid j = \sigma(left(i))\}$
$\sigma'_{right} = \{[i] \to [j] \mid j = \sigma(right(i))\}$
construct explicit relation
$A_{I_e \to x_1} = \{[e] \to [q] \mid q = \sigma'_{left}(e)\} \ \cup \ \{[e] \to [q] \mid q = \sigma'_{right}(e)\}$
locgroup(...)
$\{[i] \to [j] \mid j = \delta(i)\}$
invert(...)
$\delta^{-1} = \{[j] \to [i] \mid j = \delta(i)\}$
construct explicit relation
construct explicit relation
$\sigma''_{left} = \{[i] \to [j] \mid j = \sigma'_{left}(\delta^{-1}(i))\}$
$\sigma''_{right} = \{[i] \to [j] \mid j = \sigma'_{right}(\delta^{-1}(i))\}$
construct explicit relation
$\{[0,i] \to [1,e] \mid i = \sigma''_{left}(e)\} \ \cup \ \{[0,i] \to [1,e] \mid i = \sigma''_{right}(e)\}$
fullSparseTile(...)
$\{[x,y] \to [t] \mid t = \theta(x,y)\}$

Figure 11: The inspector dependence graph after the compile-time application of pointer update.

For the molecular dynamics example, the full sparse tiling transformation is applied after the iteration permutation called locality grouping (i.e., `locgroup()` in Figure 10). The light grey nodes in Figure 10 represent the nodes inserted into the IDG due to the locality grouping transformation described in Section 2.5. Note that the `invert()` call is inserted due to the modifications that occur to the data access functions after the locality grouping transformation (i.e., $\texttt{fx[}\sigma\texttt{[left[}e_2\texttt{]]]}$ becomes $\texttt{fx[}\sigma\texttt{[left[}\delta^{-1}\texttt{[}e_2\texttt{]]]]}$).

The white nodes in the IDG are those inserted for the full sparse tiling transformation. The sparse tiling algorithm `fullSparseTile()` performs a block partitioning of the iterations in the `e` loop for the seed partitioning and then inspects all the dependences to and from the seed space within the sub space of the full computation that is being sparse tiled. The full sparse tiling transformation implementation includes methods for updating the MapIR and IDG given input from the transformation user. Currently the user of the full sparse tiling transformation specifies what subspace of the iteration space is being sparse tiled, what seed space should be used for the seed partitioning, the transformation specification as shown in Figure 6, and which dependences are carried within the subspace being sparse tiled and have the seed partitioning subspace as a source or target.

5. Experimental Results

We experimentally compared the performance of automatically-generated inspectors and executors with hand-written ones for the moldyn and sparse matrix-vector product (spmv) benchmarks. The execution time of the generated executors should come close to the execution time of handwritten transformed code for this technology to be successful. To work toward improving the performance of the generated code, we also evaluate the effectiveness of the pointer update and guard encapsulation code-improving transformations and some additional code-improving transformations that were not incorporated into the IEGen Python prototype. Inspector execution time has less impact on application performance than that of the executors. The inspectors are executed a single time; the execution time should be within a range that can be amortized across the repeated use of the executors.

The results show that more work on improving the generated code is needed. The performance of the generated executors on the moldyn benchmark is between 5% and 60% slower than the hand-written executors. The performance of the generated inspectors ranges from 25% to 260% slower. For the spmv benchmark the slowdown in the optimized executor was around 2x. The spmv inspector was competitive until the extra work for performing optimizations for the executor was included.

Finally we observe the performance impact of the run-time reordering transformations that were implemented in the prototype. Generally, each benchmark, input file, and architecture combination requires significant tuning to determine the best combination of RTRTs to apply and how to parameterize them. Here we only tune the number of sparse tiles or cache blocks used but do not attempt

Table 1: Table of input data files used with moldyn benchmark.

Name	Num atoms	Num interactions	Average inter/atom	Footprint in MB
1TTF	50,472	9,328,136	185	75.6
3CC2	90,886	294,253	3.2	8.5
2ZV5	80,652	236,600	2.9	7.3
2ZV4	80,652	236,587	2.9	7.3
2ZUO	80,652	236,514	2.9	7.3

to find the best RTRT sequence. Our goal here is to focus on the comparison between automatically generated and hand-written inspectors and executors. However, we do note some performance wins by the RTRTs we use even when compared with Intel MKL's sparse matrix vector product implementation [59].

5.1. Experimental Methodology

We ran our experiments on a six core HP-server. Each core is an Intel Xeon CPU E5-1650 running at 3.50GHz. The 15MB L3 is shared among all cores. Each 256KB L2 is shared between 2 PUs on a single core and the L1 caches are 32KB. The operating system is Fedora release 21. MKL version 11.1.1 was used as a comparison point for the spmv experiments. The code was compiled with icc version 14.0.0 and the flags "-O3 -DNDEBUG".

A key aspect to obtaining consistent execution times for MKL was using the taskset command to pin execution of each of our executors to a single PU. Otherwise, migration between PUs was causing cache effects that led to differences in execution times of an order of magnitude or more. We also ran each executor 100× and took the average of the execution times. The machine being used was quiescent in that no one else could log into it.

5.2. The moldyn benchmark

The moldyn benchmark [60] is sparse in that there are a set of atoms and the data arrays for the atoms are accessed indirectly through index arrays that track interactions between pairs of atoms. The example in Figure 1 is a simplified version of the moldyn benchmark.

Table 1 presents the five data sets we selected for use with the moldyn benchmark. It contains the input file name, number of atoms, number of interactions, average number of interactions per atom, and footprint of the data including atoms and interactions. All of the datasets are from the Protein Data Bank [61].

5.2.1. Executor and Inspector Execution Times

Figure 12 shows the execution times for the different versions of the moldyn executor for a number of input data sets. The yellow bars all correspond to executors that have been generated by the IEGen Python prototype. The blue bars correspond to the handwritten executors. For each input file, we show four

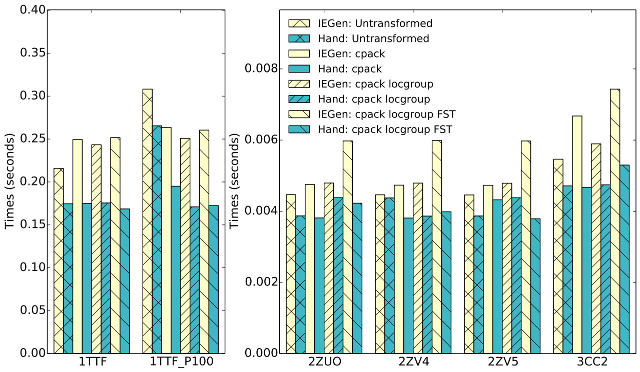

Figure 12: Executor execution times of the generated code grouped by input data file.

code versions that have been generated by the IEGen Python prototype and written by hand: untransformed, after applying the consecutive packing (cpack) data reordering, after applying consecutive packing and the locality grouping (locgroup) iteration reordering, and after applying consecutive packing, locality grouping, and full sparse tiling (FST) across the three loops within the outer time stepping loop. We also apply pointer update after consecutive packing and locality grouping, and apply the guard encapsulation optimization after FST.

This paper focuses on the performance difference between the hand-written inspectors and executors and the ones generated by the IEGen Python prototype. Figure 13 highlights the performance difference between these two by showing the execution time for the IEGen executors normalized to the time of the hand-written executors. Figure 13 shows that our generated executor code code performs no worse than 60% slower than the hand-optimized executor version. Figure 14 shows our results for the generated inspectors for the moldyn benchmark.

5.2.2. Discussion of the moldyn Inspector and Executor Results

The normalized results for the executors in Figure 13 and the inspectors in Figure 14 indicate that there is still some overhead in the generated code. We deal with some of the overhead resulting from the more general reordering algorithms by having specialized explicit relation implementations based on the relation arity as was discussed in Section 3.2. However, the generated inspector code still does not match the hand-written inspector code. One issue is that in the hand-written code, the pointer update is incorporated into the reordering algorithms because the inspector is specialized to the specific index array data structures in the benchmark. This reduces the number of traversals over the index arrays in the inspector by one.

Another issue is that all of the data dependences in the molecular dynamics benchmark are inspected in the more general IEGen full sparse tiling algorithm. In the inspector implementation that was written by hand, the inspector as-

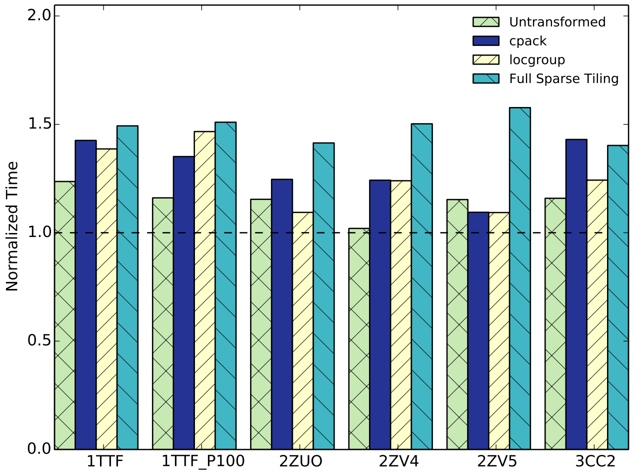

Figure 13: Each executor execution time of the generated code is normalized to the corresponding hand-optimized version, grouped by input data file.

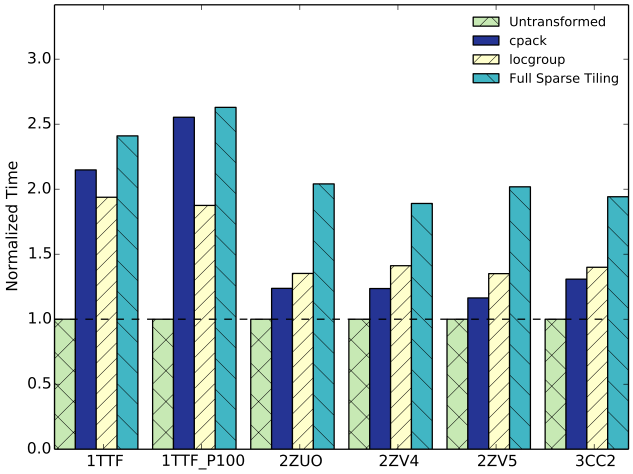

Figure 14: Each inspector execution time of the generated code is normalized to the corresponding hand-optimized version, grouped by input data file.

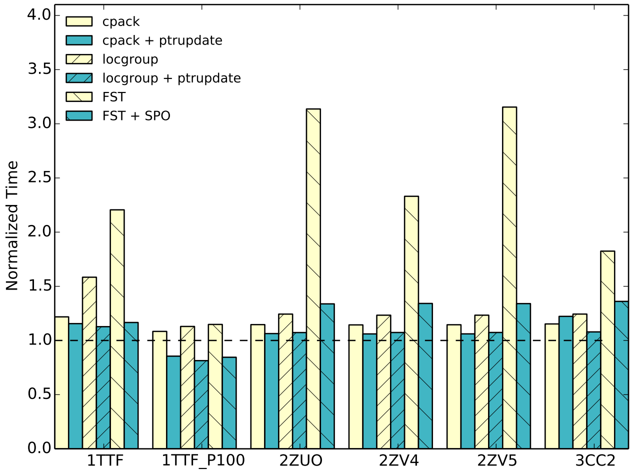

Figure 15: Executor execution times of the generated code with and without code-improving transformations: pointer update and guard encapsulation (referred to here as SPO for sparse loop optimization). Each bar is normalized to the untransformed generated version and the bars are grouped by input data file.

sumes that the dependences between loops `i` and `e` mirror the dependences between loops `e` and `i`. As such the hand-written inspector avoids a separate traversal over the dependences coming into the seed partition space and those going out of the seed partition space. Since the SPF representation of the dependences uses abstract relations, it would be possible to detect this symmetry at compile time and specialize a reordering algorithm such as full sparse tiling. This would require however that the reordering algorithms be implemented in a higher-level scripting language instead of as C run-time libraries, which is what the current prototype implementation does.

Yet another issue is that the data dependences are explicitly constructed outside of the full sparse tiling reordering algorithm and passed in as input. This requires an additional pass over index arrays that the hand-written inspectors do not need to do. This could also be solved by doing some kind of specialized code generation of the reordering algorithms.

5.2.3. Evaluation of Code-Improving Transformations

The executor and inspector results in Figures 13 and 14 already incorporate the use of the code improving transformations pointer update and guard encapsulation. Figure 15 shows the executor performance with and without the code-improving transformations. The yellow bars show the normalized execution time of various versions without code-improving transformations, and the solid blue bars are the normalized execution time with code-improving transformations. Note that the guard encapsulation is critical for executor performance. When the guard encapsulation is not used, the slowdown can be over 3×. Perform-

Table 2: Table of input data files used with SpMV benchmark.

Name	Average non-zeros/column	Num rows	Num cols	Num non-zeros	Footprint in MB
cage13	17	445,315	445,315	7,479,343	120
torso1	73	116,158	116,158	8,516,500	132
kim2r	24	456,976	456,976	11,330,020	179
nd24k	399	72,000	72,000	28,715,634	439
spal_004	143	10,203	321,696	46,168,124	707
ldoor	49	952,203	952,203	46,522,475	721

ing pointer update after the consecutive packing data reordering and locality grouping iteration reordering improves the performance of the executor slightly. For these code-improving transformations, the inspector performance actually degrades because of the extra overhead needed to actually perform the pointer update and guard encapsulation.

Figure 15 also shows that the sequence of RTRTs we selected only result in improved performance over the original code for the 1TTF_P100 dataset. That is the dataset where the atoms from the 1TTF PDB file have been 100% randomly reordered. This points to a crucial aspect of using RTRTs in that although there are many contexts where RTRTs have been found useful, finding profitable combinations depends highly on the structure of the data itself.

5.3. Sparse Matrix Vector Multiply

The sparse matrix vector multiply (SpMV) benchmark measures the time it takes to multiply a sparse matrix by a dense vector. SpMV is an important kernel in many applications [62]. There are many optimizations that are applicable to SpMV. Most of them involve some form of reordering of the non-zeros in the sparse matrix. To evaluate the IEGen Python prototype, we wrote the cache blocking transformation by hand and then specified it using SPF and generated code with IEGen. Table 2 shows the datasets we use with the SpMV benchmark. All of the sparse matrices are from the Florida Sparse Matrix collection [63].

5.3.1. Executor Execution Times

Figure 16 shows the execution times for the various SpMV executors. SpMV is typically executed using a compressed sparse row (CSR) representation, so the first bar represents a handwritten version that uses CSR. The next bar is a handwritten version using coordinate storage (COO). In the SPF, we represent computations using flat sparse data structures like COO before applying transformations. The third bar labeled IEGen COO shows the non-transformed version of the executor as generated by the IEGen Python prototype. The fourth bar shows a handwritten version of cacheblocking. This handwritten version is specialized and fused in that the cache blocking, pointer updates, data remappings, and guard encapsulation all occur within the same set of loops. The

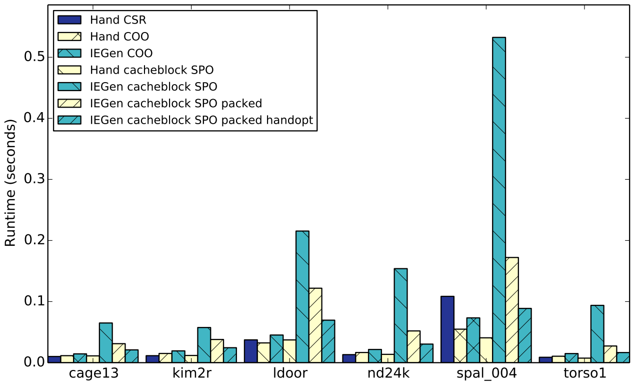

Figure 16: Executor execution times of the generated code, grouped by input data file. SPO stands for sparse loop optimization, which is guard encapsulation.

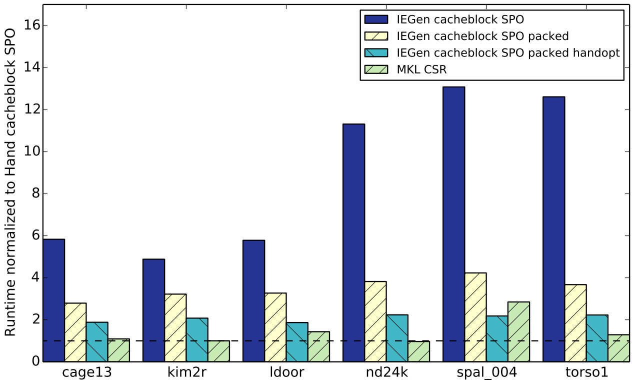

Figure 17: Executor execution times of the generated code normalized to the handwritten cache blocked version, grouped by input data file. SPO stands for sparse loop optimization, which is guard encapsulation.

IEGen cacheblock SPO and cacheblock SPO packed versions break up the specification of each of these components and enable their specification in a more general way. The last IEGen bar shows the performance of the IEGen executor after we perform some hand optimizations, which we discuss below.

We selected the sparse matrix spal_004 because [64] indicated that cache blocking should work well with this matrix. Figure 16 shows that cache blocking does perform well on this matrix, but interestingly enough coordinate storage performs just as well. The other matrices were selected at random from the Florida sparse matrix collection to provide a broad range of matrix sizes and sparsity. Cache blocking (the handwritten executor that uses cache blocking is the dashed line) also improves over the performance of MKL CSR for the cage13, ldoor, and torso matrices.

We can evaluate the code generated by the IEGen Python prototype by comparing its performance to the handwritten code even if the transformation being applied does not result in a performance improvement. Figure 17 shows the various versions of IEGen cache blocking normalized to the handwritten executor. The IEGen cacheblock SPO version performs the cache blocking and the guard encapsulation optimization, but does not reorder the non-zeros based on cache block and row. The IEGen cacheblock SPO packed version does reorder the non-zeros. The results are mixed. In all of the cases the IEGen cacheblocked versions of the executors experience a slowdown. In all cases the hand-optimized version of the IEGen code performs within 2× of the hand-written cache blocked code.

5.3.2. Code-Improving Transformations Applied by Hand

The IEGen cacheblock SPO packed hand-optimized version incorporates some hand optimizations to the inspector and the executor. For the executor, we know that since the non-zeros have been packed based on their cache block and row that the innermost loop only needs the guard encapsulation data structure to count the number of non-zeros per cache block and row. We modify the innermost loop so that the index into the non-zero values and column arrays is sequential. The handwritten cache blocked version already takes advantage of this.

The other optimization that could be easily incorporated into IEGen is the realization that the `cb` and `row` arrays are used in the executor code after the guard encapsulation. Therefore, it was not necessary to perform pointer update on them in the inspector.

5.3.3. Inspector Execution Times

Figure 18 shows the execution time of the inspectors, and Figure 19 shows those execution times normalized to the handwritten cache blocking inspector. Even the hand-optimized inspector is sometimes more than twice the execution time of the handwritten inspector. This suggests that more optimizations within the IEGen generated code are needed. This time difference is probably due to some of the excess memory and work needed to explicitly pass the mapping of nonzeros to cache blocks to the data packing algorithm and the

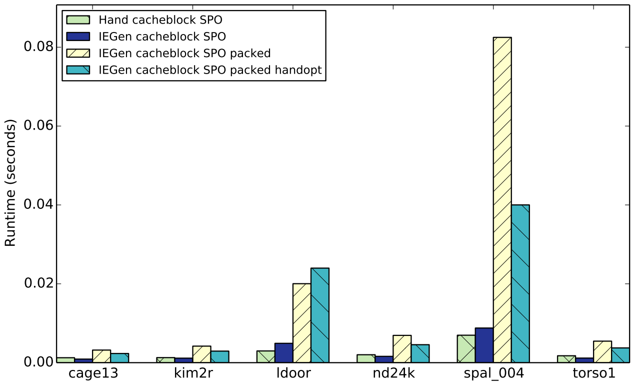

Figure 18: Inspector execution times of the generated code, grouped by input data file. SPO stands for sparse loop optimization, which is guard encapsulation.

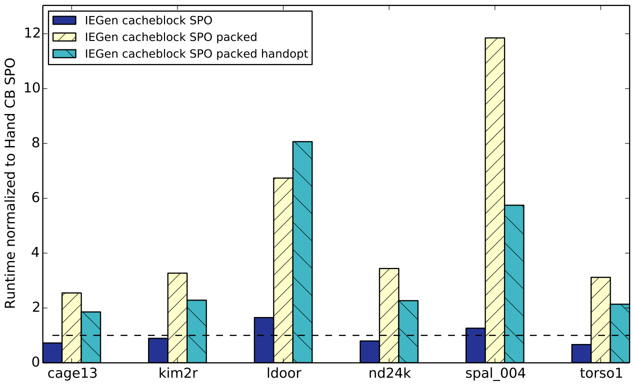

Figure 19: Inspector execution times of the generated code normalized to the handwritten cacheblock version, grouped by input data file. SPO stands for sparse loop optimization, which is guard encapsulation.

guard encapsulation pieces. It should be possible to leverage the abstract set and relation descriptions to fuse some of this work at compile time when the reordering algorithms themselves are specified in a higher level language instead of C run-time library routines.

6. Related Work

In this section, we review existing Runtime Reordering Transformations (RTRTs) and indicate which of the existing RTRTs can be expressed within the sparse polyhedral framework (SPF). To organize the discussion, we place RTRTs into categories based on whether they permute data or loop iterations, or increase the dimensionality of a data array or loop (embeddings, or groupings). We also overview some ongoing development of various RTRTs for the sparse matrix vector benchmark.

6.1. Data and Iteration Permutation Reorderings

A number of data and iteration permutation reorderings have been developed in the context of loops with no inter-iteration dependences or only reduction dependences. The goal of these data and iteration permutations is to improve the data locality within an irregular loop. Such run-time reordering transformations inspect access functions (the mapping of iterations to data) to determine a better data or iteration permutation. The most common approach for using these RTRTs is to perform a data permutation and then an iteration permutation [30]. For example,

SPF can express any one-dimensional loop permutation or data permutation as an abstract relation where the output tuple variable is specified as equivalent to the output value of an uninterpreted function that represents the reordering,

$$\{[x] \rightarrow [y] \mid y = f(x)\}.$$

Permutation RTRTs that SPF can represent include Cuthill-McKee [19], Reverse Cuthill-McKee [65], breadth-first [23], Sloan [26], recursive coordinate bisection [66], consecutive packing [24], reordering based on graph partitioning [21, 30], hybrid techniques based on graph partitioning and another heuristic within the partition [23, 67], reordering based on space-filling curves [29], lexicographical grouping or sorting [20, 24, 26], and hyper-breadth-first [68]. The reordering algorithms that depend on a mapping of data indices to simulation space coordinate data (e.g., recursive coordinate bisection [66] and space-filling curves [29]) will require additional input be provided to the inspector, but this input can be expressed as an abstract relation.

The SPF can also express loop and data permutation transformations such as array alignment and iteration alignment that are performed to localize memory accesses occurring in loops other than the loop where an initial data or iteration permutation occurred.

6.2. Data and Iteration Embedding Reorderings

A *data embedding reordering* is a transformation that introduces an additional dimension to an array. Smashing [69] is an example of a data embedding reordering that folds regular data spaces to remove non-uniform dependences in regular computations. Smashing can be expressed in the SPF as affine transformations on the data space.

An *iteration embedding reordering* is a transformation that introduces another loop into a computation to iterate over groups of iteration points in some way. Iteration embeddings are used to improve data locality and/or parallelize irregular computations.

In the context of improving data locality, an example transformation is bucket-tiling [25], where iterations are placed into buckets based on the range of data accessed within the iteration. The cache blocking provided by OSKI [27, 62] used within the context of sparse matrix vector multiplication is another grouping data locality improving transformation. Both bucket tiling and cache blocking can be expressed within the SPF.

The sparse tiling transformations, unstructured cache blocking [50], full sparse tiling [28, 49], and communication avoiding rescheduling[70] improve temporal data locality in computations and also can be used to create coarse-grain parallelism by grouping iterations across iterations in an outer loop or between loops. The sparse tiling transformations are expressible within the SPF and the IEGen Python prototype can generate inspectors and executors for the serial version of these transformations.

In the context of parallelizing irregular applications, gather/scatter parallelism is commonly used in irregular applications where the programmer has specified the data decomposition for a distributed array [71, 72, 73, 74, 75]. Typically there is language and compilation support for data distribution specifications, parallel loops, and reductions, which involves generating code with calls to the appropriate inspector, scheduling, and gather/scatter functions in a run-time library such as CHAOS [76]. The sparse polyhedral framework (SPF) and IEGen runtime build on these ideas with the key extensions being that many more transformation types can be specified with the SPF, and the transformations being applied can be specified as well as the original computation. Although the SPF enables the specification of parallel schedules, the IEGen Python prototype does not generate parallel code.

When parallelizing irregular loops with loop-carried dependences, an inspector must determine the dependences at run-time before rescheduling the loop. One approach is to dynamically schedule iterations into wavefronts such that all of the iterations within one wavefront may be executed in parallel. In [77], Rauchwerger surveys various techniques for dynamically scheduling iterations into wavefronts such that all of the iterations within one wavefront may be executed in parallel. An inspector for detecting partial parallelism inspects all the dependences for a loop, and places iterations into wavefronts. The SPF can express partial parallelism transformations, but again the IEGen code generator does not yet generate parallel code.

6.3. Sparse Matrix Vector Multiplication

Sparse matrix vector multiplication is an important kernel that is the performance bottleneck in many large scale scientific applications. As such, techniques for improving its performance have been studied in the context of multicore architectures [78, 79], GPUs [80, 81], and Xeon Phi [82, 83]. Many of these techniques include an inspector phase where reordering or reorganization of the sparse matrix non-zeros occurred. Other techniques include various parallelizations and vectorization. The sparse polyhedral model is focused on how the application of reordering techniques can be automated in a compiler in a way that various reorderings can be composed. We have started incorporating some of the concepts of the sparse polyhedral framework for specifying loop transformations and using those loop transformations to transform code to operate on different sparse matrix data structure formats [84, 85].

7. Conclusions

The performance optimization process for irregular/sparse scientific applications has generally been hand-coded and/or supported with libraries, and typically involves using inspector/executor strategies to implement various Run-Time Reordering Transformations (RTRTs). In this paper, we present the Sparse Polyhedral Framework (SPF) for specifying irregular/sparse computations and RTRTs on those computations. We show how to represent inspectors and executors at compile time with the Inspector Dependence Graph (IDG) and Mapping Intermediate Representation (MapIR), manipulate those representations to show the effect of the RTRTs being applied, and then generate the inspector and executor code. Additionally we present code-improving transformations that do not reorder data or computation, but perform other transformations such as collapsing nested index arrays to improve the inspector and executor performance. Finally, we show experimental results that indicate the generated inspectors and executor still require some optimizations in the generated code to compete with the performance of handwritten code, and we explain what remaining gaps exist.

8. Acknowledgements

We thank Jon Roelofs for his implementation of the IEGenCC tool, which converts C programs into the specification format IEGen expects as input. We thank Christopher Krieger, Andrew Stone, Tomofumi Yuki, and anonymous reviewers for their careful reading and suggestions. This work was sponsored by NSF CAREER grant CCF-0746693, DOE Early Career grant DE-SC3956, the CSCAPES institute DOE grant 7F-00323, and the CACHE project DOE grant DE-SC04030.

[1] V. E. Taylor, R. L. Stevens, K. E. Arnold, Parallel molecular dynamics: implications for massively parallel machines, Journal of Parallel and Distributed Computing 45 (2) (1997) 166–175.

[2] S. Goedecker, A. Hoisie, Performance Optimization of Numerically Intensive Code, SIAM, Philadelphia, PA, USA, 2001.

[3] P. Feautrier, Some efficient solutions to the affine scheduling problem. part II. multidimensional time, International Journal of Parallel Programming 21 (6) (1992) 389–420.

[4] L.-C. Lu, A unified framework for systematic loop transformations, in: Proceedings of the 3rd Annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), ACM, New York, NY, USA, 1991, pp. 28–38.

[5] V. Sarkar, R. Thekkath, A general framework for iteration-reordering loop transformations, in: C. W. Fraser (Ed.), Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM, New York, NY, USA, 1992, pp. 175–187.

[6] W. Li, K. Pingali, A singular loop transformation framework based on non-singular matrices, International Journal on Parallel Processing 22 (2) (1994) 183–205.

[7] S. Carr, K. S. McKinley, C.-W. Tseng, Compiler optimizations for improving data locality, in: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM, New York, NY, USA, 1994, pp. 252–262.

[8] W. Kelly, W. Pugh, A unifying framework for iteration reordering transformations, Technical Report CS-TR-3430, University of Maryland, College Park (February 1995).

[9] I. Kodukula, K. Pingali, Transformations for imperfectly nested loops, in: Proceedings of the ACM/IEEE Conference on Supercomputing, IEEE Computer Society, Washington, DC, USA, 1996.

[10] M. E. Wolf, D. E. Maydan, D.-K. Chen, Combining loop transformations considering caches and scheduling, in: Proceedings of the 29th Annual International Symposium on Microarchitecture, IEEE Computer Society Press, Los Alamitos, CA, USA, 1996, pp. 274–286.

[11] M. Kandemir, A. Choudhary, J. Ramanujam, P. Banerjee, Improving locality using loop and data transformations in an integrated framework, in: Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO), IEEE Computer Society Press, Los Alamitos, CA, USA, 1998, pp. 285–296.

[12] M. Cierniak, W. Li, Unifying data and control transformations for distributed shared-memory machines, in: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation, PLDI '95, ACM, New York, NY, USA, 1995, pp. 205–217.

doi:10.1145/207110.207145.
URL http://doi.acm.org/10.1145/207110.207145

[13] W. Thies, F. Vivien, J. Sheldon, S. Amarasinghe, A unified framework for schedule and storage optimization, in: C. Norris, J. J. B. Fenwick (Eds.), Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vol. 36.5 of ACM SIGPLAN Notices, ACM, New York, NY, USA, 2001, pp. 232–242.

[14] M. E. Wolf, M. S. Lam, Loop transformation theory and an algorithm to maximize parallelism, IEEE Transactions on Parallel and Distributed Systems 2 (4) (1991) 452–471.

[15] A. Cohen, S. Donadio, M.-J. Garzaran, C. Herrmann, O. Kiselyov, D. Padua, In search of a program generator to implement generic transformations for high-performance computing, Sci. Comput. Program. 62 (1) (2006) 25–46.

[16] U. Bondhugula, A. Hartono, J. Ramanujam, P. Sadayappan, A practical automatic polyhedral program optimization system, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM, New York, NY, USA, 2008.

[17] F. Quilleré, S. Rajopadhye, Optimizing memory usage in the polyhedral model, ACM Transactions on Programming Languages and Systems 22 (5) (2000) 773–815.

[18] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, C. Bastoul, The polyhedral model is more widely applicable than you think, in: Compiler Construction, Vol. LNCS 6011, Springer-Verlag, Berlin, Heidelberg, 2010.

[19] E. Cuthill, J. McKee, Reducing the bandwidth of sparse symmetric matrices, in: Proceedings of the 24th National Conference ACM, ACM, New York, NY, USA, 1969, pp. 157–172.

[20] R. Das, D. Mavriplis, J. Saltz, S. Gupta, R. Ponnusamy, The design and implementation of a parallel unstructured euler solver using software primitives, AIAA Journal 32 (1992) 489–496.

[21] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, J. Hennessy, Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-Hut, fast multipole, and radiosity, Journal of Parallel and Distributed Computing 27 (2) (1995) 118–141.

[22] J. Saltz, C. Chang, G. Edjlali, Y.-S. Hwang, B. Moon, R. Ponnusamy, S. Sharma, A. Sussman, M. Uysal, G. Agrawal, R. Das, P. Havlak, Programming irregular applications: Runtime support, compilation and tools, in: M. V. Zelkowitz (Ed.), Emphasizing Parallel Programming Techniques, Vol. 45 of Advances in Computers, Elsevier, 1997, pp. 105 –

153. doi:10.1016/S0065-2458(08)60707-X.
URL http://www.sciencedirect.com/science/article/pii/S006524580860707X

[23] I. Al-Furaih, S. Ranka, Memory hierarchy management for iterative graph structures, in: Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP-98), IEEE Computer Society, Los Alamitos, CA, USA, 1998, pp. 298–302.

[24] C. Ding, K. Kennedy, Improving cache performance in dynamic applications through data and computation reorganization at run time, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, 1999, pp. 229–241.

[25] N. Mitchell, L. Carter, J. Ferrante, Localizing non-affine array references, in: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), IEEE Computer Society, Los Alamitos, CA, USA, 1999, pp. 192–202.

[26] J. Fu, A. Pothen, D. Mavriplis, S. Ye, On the memory system performance of sparse algorithms, in: Proceedings of the Eighth International Workshop on Solving Irregularly Structured Problems in Parallel, IEEE Computer Society, Los Alamitos, CA, USA, 2001.

[27] E.-J. Im, K. Yelick, Optimizing sparse matrix computations for register reuse in sparsity, in: V.N.Alexandrov, J. Dongarra, C.J.K.Tan (Eds.), Proceedings of the International Conference on Computational Science (ICCS), Vol. 2073 of Lecture Notes in Computer Science, Springer, Berlin / Heidelberg, 2001, pp. 127–136.

[28] M. M. Strout, L. Carter, J. Ferrante, Rescheduling for locality in sparse matrix computations, in: V. N. Alexandrov, J. J. Dongarra, C. J. K. Tan (Eds.), Proceedings of the International Conference on Computational Science (ICCS), LNCS 2073, Springer, Berlin / Heidelberg, 2001.

[29] J. Mellor-Crummey, D. Whalley, K. Kennedy, Improving memory hierarchy performance for irregular applications using data and computation reorderings, International Journal of Parallel Programming 29 (3) (2001) 217–247.

[30] H. Han, C.-W. Tseng, Exploiting locality for irregular scientific codes, IEEE Transactions on Parallel and Distributed Systems 17 (7) (2006) 606–618.

[31] R. Das, M. Uysal, J. Saltz, Y.-S. S. Hwang, Communication optimizations for irregular scientific computations on distributed memory architectures, Journal of Parallel and Distributed Computing 22 (3) (1994) 462–478.
URL citeseer.nj.nec.com/das93communication.html

[32] S. D. Sharma, R. Ponnusamy, B. Moon, Y.-S. Hwang, R. Das, J. Saltz, Run-time and compile-time support for adaptive irregular problems, in: Proceedings of the Conference on Supercomputing, IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[33] J. Wu, R. Das, J. H. Saltz, H. Berryman, S. Hiranandani, Distributed memory compiler design for sparse problems, IEEE Transactions on Computers 44 (6) (1995) 737–754.

[34] M. M. Strout, A. LaMielle, L. Carter, J. Ferrante, B. Kreaseck, C. Olschanowsky, An approach for code generation in the sparse polyhedral framework, Tech. Rep. CS-13-109, Colorado State University (December 2013).

[35] U. Banerjee, Unimodular transformations of double loops, in: A. Nicolau, D. Gelernter, T. Gross, D. Padua (Eds.), Advances in Languages and Compilers for Parallel Computing, MIT Press, Cambridge, MA, USA, 1990, pp. 192–219.

[36] M. E. Wolf, M. S. Lam, A data locality optimizing algorithm, in: Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM, New York, NY, USA, 1991, pp. 30–44.

[37] P. Feautrier, Parametric integer programming, RAIRO Recherche Op'erationnelle 22.

[38] C. Bastoul, A. Cohen, A. Girbal, S. Sharma, O. Temam, Putting polyhedral loop transformations to work, in: In Workshop on Languages and Compilers for Parallel Computing (LCPC), LNCS 2958, Springer, Berlin / Heidelberg, 2003, pp. 209–225.

[39] M. M. Strout, L. Carter, J. Ferrante, Compile-time composition of run-time data and iteration reorderings, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM, New York, NY, USA, 2003.

[40] B. Pugh, D. Wonnacott, Nonlinear array dependence analysis, Tech. Rep. CS-TR-3372, Dept. of Computer Science, Univ. of Maryland (November 1994).

[41] M. M. Strout, G. George, C. Olschanowsky, Set and relation manipulation for the sparse polyhedral framework, in: Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing (LCPC), 2012.

[42] N. Ahmed, N. Mateev, K. Pingali, Synthesizing transformations for locality enhancement of imperfectly-nested loop nests, in: Conference Proceedings of the 2000 International Conference on Supercomputing, Kluwer Academic Publishers, Norwell, MA, USA, 2000, pp. 141–152.

[43] C. Bastoul, CLooG: A loop generator for scanning polyhedra, edition 2.1, for CLooG 0.16.0, http://www.bastoul.net/cloog/pages/download/count.php3?url=./cloog.pdf (October 15th 2007).

[44] M. M. Strout, L. Carter, J. Ferrante, Proof of correctness for sparse tiling of Gauss-Seidel, Tech. Rep. CS2003-0741, University of California, San Diego (2003).

[45] M. Norrish, M. M. Strout, An approach for proving the correctness of inspector/executor transformations, in: Proceedings of the 27th International Workshop on Languages and Compilers for Parallel Computing (LCPC), 2014.

[46] A. Venkat, M. Shantharam, M. Hall, M. M. Strout, Non-affine extensions to polyhedral code generation, in: To be published in: Proceedings of International Symposium on Code Generation and Optimization CGO), 2014.

[47] W. Kelly, W. Pugh, Finding legal reordering transformations using mappings, in: Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing, Vol. 892, Springer-Verlag, London, UK, 1995, pp. 107–124.

[48] M. M. Strout, L. Carter, J. Ferrante, Proof of correctness for sparse tiling of gauss-seidel, Tech. rep., UCSD Department of Computer Science and Engineering, Technical Report #CS2003-0741 (April 2003).

[49] M. M. Strout, L. Carter, J. Ferrante, B. Kreaseck, Sparse tiling for stationary iterative methods, International Journal of High Performance Computing Applications 18 (1) (2004) 95–114.

[50] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, C. Weiss, Cache Optimization for Structured and Unstructured Grid Multigrid, Electronic Transaction on Numerical Analysis 10 (2000) 21–40.

[51] J. Demmel, M. Hoemmen, M. Mohiyuddin, K. Yelick, Avoiding communication in sparse matrix computations, in: Proceedings of International Parallel and Distributed Processing Symposium (IPDPS), IEEE Computer Society, Los Alamitos, CA, USA, 2008.

[52] A. J. C. Bik, H. A. G. Wijshoff, Automatic data structure selection and transformation for sparse matrix computations, IEEE Trans. Parallel Distrib. Syst. 7 (2) (1996) 109–126.

[53] G. Rudy, C. Chen, M. Hall, M. M. Khan, J. Chame, Using a programming language interface to describe GPGPU optimization and code generation, in: The 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC), 2010.

[54] T. Brandes, The importance of direct dependences for automatic parallelism, in: Proceedings of the International Conference on Supercomputing, ACM, New York, NY, USA, 1988, pp. 407–417.

[55] P. Feautrier, Dataflow analysis of array and scalar references, International Journal of Parallel Programming 20 (1) (1991) 23–53.
URL `http://citeseer.ist.psu.edu/feautrier91dataflow.html`

[56] D. E. Maydan, J. L. Hennessy, M. S. Lam, Efficient and exact data dependence analysis, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, New York, NY, USA, 1991, pp. 1–14.

[57] W. Pugh, D. Wonnacott, An exact method for analysis of value-based array data dependences, in: U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), Proceedings of the Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing (LCPC), Vol. 768 of Lecture Notes in Computer Science, Springer-Verlag, London, UK, 1993.

[58] Y. Lin, D. Padua, Compiler analysis of irregular memory accesses, SIGPLAN Notices 35 (5) (2000) 157–168.

[59] I. D. Zone, Intel math kernel library, https://software.intel.com/en-us/articles/intel-math-kernel-library-documentation (2015).

[60] R. Ponnusamy, Y.-S. Hwang, R. Das, J. Saltz, A. Choudhary, G. Fox, Supporting irregular distributions in Fortran 90D/HPF compilers, Tech. Rep. UMIACS-TR-94-57.1, University of Maryland at College Park, College Park, MD, USA (1994).

[61] San Diego Supercomputer Center, Protein Data Bank, http://www.rcsb.org/pdb/home/home.do (2010).

[62] R. Vuduc, J. W. Demmel, K. A. Yelick, OSKI: A library of automatically tuned sparse matrix kernels, in: Proceedings of SciDAC 2005, Journal of Physics: Conference Series, Institute of Physics Publishing, San Francisco, CA, USA, 2005.

[63] T. Davis, University of Florida sparse matrix collection, http://www.cise.ufl.edu/research/sparse/matrices/ (2010).

[64] R. Nishtala, R. W. Vuduc, J. W. Demmel, K. A. Yelick, When cache blocking of sparse matrix vector multiply works and why, Applicable Algebra in Engineering, Communication and Computing 18 (3) (2007) 297–311.

[65] J. Liu, A. Sherman, Comparative analysis of the Cuthill-Mckee and the reverse Cuthill-Mckee ordering algorithms for sparse matrices, SIAM Journal of Numerical Analysis 13 (2) (1976) 198–213.

[66] R. D. Williams, Performance of dynamic load balancing algorithms for unstructured mesh calculations, Concurrency: Practice and Experience 3 (5) (1991) 457–481.

[67] M. M. Strout, N. Osheim, D. Rostron, P. D. Hovland, A. Pothen, Evaluation of hierarchical mesh reorderings, in: Proceedings of the International Conference on Computational Science (ICCS), no. 5544 in LNCS, Springer, Berlin / Heidelberg, 2009.

[68] M. M. Strout, P. D. Hovland, Metrics and models for reordering transformations, in: Proceedings of the The Second ACM SIGPLAN Workshop on Memory System Performance (MSP), ACM, New York, NY, USA, 2004, pp. 23–34.

[69] N. Osheim, M. M. Strout, D. Rostron, S. Rajopadhye, Smashing: Folding space to tile through time, in: The Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC), Vol. LNCS 5335, Springer, Berlin / Heidelberg, 2008.

[70] M. Mohiyuddin, M. Hoemmen, J. Demmel, K. Yelick, Minimizing communication in sparse matrix solvers, in: Supercomputing, ACM, New York, NY, USA, 2009.

[71] J. Saltz, K. Crowley, R. Mirchandaney, H. Berryman, Run-time scheduling and execution of loops on message passing machines, Journal of Parallel and Distributed Computing 8 (4) (1990) 303–312.

[72] C. Koelbel, P. Mehrotra, Compiling global name-space parallel loops for distributed execution, Parallel and Distributed Systems, IEEE Transactions on 2 (4) (1991) 440–451.

[73] S. Hiranandani, K. Kennedy, C. wen Tseng, Compiling fortran D for MIMD distributed-memory machines, Communications of the ACM 35 (8) (1992) 66–80.

[74] R. von Hanxleden, K. Kennedy, C. H. Koelbel, R. Das, J. H. Saltz, Compiler analysis for irregular problems in Fortran D, in: In Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC), no. 757 in LNCS, Springer-Verlag, London, UK, 1992, pp. 97–111.

[75] B. Chapman, H. Zima, P. Mehrotra, Extending HPF for advanced data-parallel applications, IEEE Parallel Distrib. Technol. 2 (3) (1994) 59–70.

[76] R. Ponnusamy, J. Saltz, A. Choudhary, Runtime compilation techniques for data partitioning and communication schedule reuse, in: Proceedings of the ACM/IEEE Conference on Supercomputing, ACM Press, New York, NY, USA, 1993, pp. 361–370.

[77] L. Rauchwerger, Run-time parallelization: Its time has come, Parallel Computing 24 (3–4) (1998) 527–556.

[78] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel, Optimization of sparse matrix-vector multiplication on emerging multicore platforms, in: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, ACM, New York, NY, USA, 2007, pp. 1–12.

[79] A. Buluç, S. Williams, L. Oliker, J. Demmel, Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication, in: Proc. IPDPS, 2011.

[80] N. Bell, M. Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, in: SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, New York, NY, USA, 2009, pp. 1–11.

[81] A. Rafique, G. Constantinides, N. Kapre, Communication optimization of iterative sparse matrix-vector multiply on gpus and fpgas, Parallel and Distributed Systems, IEEE Transactions on PP (99) (2014) 1–1.

[82] E. Saule, K. Kaya, Ü. V. Çatalyürek, Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi, in: Proc of the 10th Int'l Conf. on Parallel Processing and Applied Mathematics (PPAM), 2013, p. 10.

[83] X. Liu, M. Smelyanskiy, E. Chow, P. Dubey, Efficient sparse matrix-vector multiplication on x86-based many-core processors, in: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS '13, ACM, New York, NY, USA, 2013, pp. 273–282.

[84] A. Venkat, M. Shantharam, M. Hall, M. M. Strout, Non-affine extensions to polyhedral code generation, in: In International Symposium on Code Generation and Optimization (CGO), 2014.

[85] A. Venkat, M. Hall, M. M. Strout, Loop and data transformations for sparse matrix code, in: In Programming Languages Design and Implementation (PLDI), 2015.