Moldyn (M2) FST Example

Michelle Mills Strout

June 12, 2009

We present the steps for automatically generating a composed inspector and the corresponding executor for a simplified version of the moldyn benchmark. Portions of this example can be used throughout the RTRT journal paper.

Figure 1 shows a computation with indirect memory references written in the C programming language. There is an outer time-stepping loop using iterator `s` and indirect memory references to the data arrays `fx` and `x` using the index arrays `inter1` and `inter2`.

To improve the performance of this computation, we plan to reorder/permute the `fx` and `x` data arrays to improve spatial data locality, reorder/permute the iterations of the `ii` loop to improve temporal locality, and then sparse tile across the `i` and `ii` loops to improve temporal locality between the `i` and `ii` loops.

We present the following details for transforming and generating the inspector and executor for the example in Figure 1:

- How the user specifies the computation in the Sparse Polyhedral Framework (SPF).
- How the transformation writer specifies possible run-time reordering transformations and provides run-time library support for each RTRT.
- How the user specifies a sequence of RTRTs to apply to the example computation.
- How we automatically generate a composed inspector and executor to implement the sequence of RTRTs.

1 Specifying the Computation in SPF

There is a python interface for specifying the computations and a preliminary tool that parses annotated C code into the python code specification. For this example, we describe the pieces of the computation that need to be specified and provide examples of how the specifications are done using the python interface.

The computation is organized in an intermediate representation called the Mapping Intermediate Representation (MapIR). The term mapping derives from

```
  for (s=0; s<T; s++) {
        for (i=0; i<N; i++) {
S1:     x[i] = fx[i]*1.25;
        }
    for (ii=0; ii<n_inter; ii++) {
S2:     fx[inter1[ii]] += x[inter1[ii]] - x[inter2[ii]];
S3:     fx[inter2[ii]] += x[inter1[ii]] - x[inter2[ii]];
    }
  }
```

Figure 1:

the main component of the specification being integer tuple mappings, or relations, between computation and computation, between computation and data, etc. First an instance of the MapIR is constructed.

```
moldyn_spec=MapIR()
```

The next step is to specify all of the symbolic constants in the computation. Symbolic constants are variables whose value does not change during the course of the computation. For Figure 1, the symbolic constants are `T`, `N`, and `n_inter`.

```
moldyn_spec.add_symbolic(name='T')
moldyn_spec.add_symbolic(name='N')
moldyn_spec.add_symbolic(name='n_inter')
```

After specifying the symbolic constants, we specify the data and index arrays. The data and index arrays are specified in terms of their name and a set specification for their data space. The data space bounds can be affine functions of the previously declared symbolic constants. For an index array, their data space is equivalent to their input bounds. The output bounds for an index array specify the space of index values that the index array could contain at runtime. Below we only show one data array example and one index array example.

```
moldyn_spec.add_data_array(
    name='x',
    bounds='{[k]: 0<=k && k<N}')

moldyn_spec.add_index_array(
    name='inter1',
    input_bounds='{[k]: 0<=k && k<n_inter}',
    output_bounds='{[k]: 0<=k && k<N}')
```

The next step is to specify each of the statements and information about how each statement accesses the data arrays. The statements are given names to enable later association with access relations. [ALAN: since we have the name for the access relation, this is not really necessary. However, it might be handy for debugging purposes, so let's keep it for now.] The statement is specified as a string with special substrings `%(a#)` that indicate the data array memory accesses in the statement. The original iteration space for the statement must be specified (`iter_space`), along with a scheduling function (`scatter`) that maps each point in that statements original iteration space to a full iteration space shared by all of the statements. The assumed schedule is that the instances of each of the statements will be executed in lexicographical order in the shared, full iteration space. The `iter_to_data` parameter mathematically describes the access relation between the original iteration space for the statement to the target data array (`data_array`).

```
moldyn_spec.add_statement(
    name='S1',
    text='x[%(a1)s] = fx[%(a2)s] * 1.25;',
    iter_space='{[s,i]: 0<=s && s<T && 0<=i && i<N}',
    scatter='{[s,i]->[c0,s,c1,i,c2]: c0=0 && c1=0 && c2=0}')

moldyn_spec.add_access_relation(
    statement_name='S1',
    name='a1',
    data_array='x',
    iter_to_data='{[s,i]->[i]}')
```

The final step is to specify the data dependences. [I think we should specify the data dependences between statements and between iterations of those statements original iteration space.] FIXME: how do we specify data dependences.

1.1 Computation Summary After Initial Specification

Each statement has an original iteration space and a scheduling function that maps each point in the original iteration space to a shared iteration space with all the other statements. 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. Therefore, an interactive tool that enables using RTRTs should show the user the initial full iteration space specification and the full iteration space specification after any iteration reordering transformations have been applied. The full iteration space is computed by applying the scheduling functions to each statement and then taking the union of the resulting sets.

A related issue is that the access relations as specified by the user map points in the original iteration space for a statement into the data space being accessed. To reflect the iteration reordering transformations in the access relations, all of the original access relation specifications are modified automatically so that they map points in the full iteration space to the data space being accessed. We do this by applying each access relation to the inverse of the original scheduling function for the associated statement. (We talked about this on 1/9/09).

For this example, the computation's initial specification can be presented as follows:

```
// full iteration space:
//    { [0, s, 0, i, 0 ] : 0<=s && s<T && 0<=i && i<N }
//    union { [0, s, 1, ii, x ] : 0<=s && s<T && 0<=ii && ii<n_inter && 0>=x && x<=1 }
  for (s=0; s<T; s++) {
    for (i=0; i<N; i++) {
S1:     x[i] = fx[i]*1.25;
    }
    for (ii=0; ii<n_inter; ii++) {
S2:     fx[inter1[ii]] += x[inter1[ii]] - x[inter2[ii]];
S3:     fx[inter2[ii]] += x[inter1[ii]] - x[inter2[ii]];
    }
  }
// data dependences:
//  D_S1_to_S2 =  {[0,s,0,i,0] -> [0,s,1,ii,0] : i = inter1(ii) }
//            union  {[0,s,0,i,0] -> [0,s,1,ii,0] : i = inter2(ii) }
//  D_S1_to_S3 =  {[0,s,0,i,0] -> [0,s,1,ii,1] : i = inter1(ii) }
//            union  {[0,s,0,i,0] -> [0,s,1,ii,1] : i = inter2(ii) }
//  D_S2_to_S1 = { [0,s1,1,ii,0] -> [0,s2,0,i,0] : s2 = s1 + 1 && i = inter1(ii) }
//            union  {[0,s1,1,ii,0] -> [0,s2,0,i,0] : s2 = s1 + 1 && i = inter2(ii) }
//  D_S3_to_S1 =  { [0,s1,1,ii,1] -> [0,s2,0,i,0] : s2 = s1 + 1 && i = inter1(ii) }
//            union  {[0,s1,1,ii,1] -> [0,s2,0,i,0] : s2 = s1 + 1 && i = inter2(ii) }
```

2 Transformation Writer Specifying RTRTs

All RTRTs include compile time components and runtime components. The compile time component requires that the user specify what data array(s) or iteration subspace(s) are to be transformed using which explicit relations that will be available at runtime. The RTRT subclass consists of methods that compute the input explicit relation *specifications* at compile time. The RTRT also includes methods that modify statement access relations and/or scheduling functions.

The code generator will generate code that computes the explicit relations needed as input to the various reordering algorithms. The reordering algorithms are called explicit relation generators, because their output is also an explicit relation.

The reordering algorithms are written by hand and are part of the run-time library support for RTRTs.

3 Usage of RTRTs

Specifications for data permutation.

```
    moldyn_spec.add_transformation(
        iegen.trans.DataPermuteTrans,
        name='cpack',
        reordering_name='sigma',
        data_arrays=['x','fx'],
        iter_sub_space_relation='{[c0,s,c1,i,c2]->[i] : c1=1}',
        target_data_array='x',
        erg_func_name='ERG_cpack')


    moldyn_spec.add_transformation(
        iegen.trans.IterPermuteTrans,
        name='lexmin',
        reordering_name='delta',
        iter_sub_space_relation='{[c0, s, c1, ii, c2] -> [ ii ] : c1=1}',
        target_data_arrays=['x','fx'],
        erg_func_name='ERG_lexmin')
```

The more intuitive specification requires the following information (information that the user must provide as parameters to IterPermuteTrans):

- The relation between the full iteration space and the iteration subspace that is being permuted. (In DataPermuteTrans something similar is called iter_sub_space_relation). For this example, we have

$$\{[c0, s, c1, ii, c2] \to [c1, ii] \mid c1 = 1\}$$

- The iteration permutation transformation assumes that it will be permuting an iteration space based on how that iteration space accesses a set of data spaces. Therefore we need a set of target data arrays.
- Finally we need the name of the reordering algorithm, or ERG, that the transformation should use.

4 Automatic Generation of Inspector and Executor

After processing each RTRT, the IDG will be extended to represent components in the inspector and the access relations and/or scheduling functions associated with various statements will be modified to represent the effect of the transformation on the executor.

How the list of RTRTs is converted into an IDG and modifies the MapIR.

4.1 Data permutation

Assume we first apply a data reordering transformation to the data arrays `x` and `fx`.

$$R_{x \to x'} = \{[k] \to [j] \mid j = sigma(k)\}$$

The above is the mathematical description of the effect the transformation will have on the dataspace x. The transformation writer creates a subclass that provides an interface for the user to specify the transformation (see specification in Section 3) and that implements the following:

- DataPermuteRTRT.calc_input generates unioned access relation for all statements in iteration sub space. In Figure 2 the create access relation is labeled ER_1.
- The DataPermuteRTRT.calc_output routine will generate an ER node representing the permutation sigma. [Alan: where is map of name sigma to that ER node?]
- The update_mapIR method in general modifies any statement scheduling functions or access relations that are affected by the transformation. For the example data reordering, any access relations targeting the data arrays being reordered need to be modified by composing the data reordering relation $R_{x \to x'}$ with each affected access relation.
- The update_IDG creates nodes in the inspector dependence graph (IDG) for inputs to the reordering algorithm (e.g. ER_1), the reordering algorithm (e.g. ERG_cpack), and the output of the reordering algorithm (e.g. ER_sigma). Nodes for the data arrays being reordered and the generic reordering algorithm are also added. In the IDG, the reordered data arrays have the number one appended to indicate that there is actually an output array. In the implementation, the reordered array is copied back into the original array. [This is probably too much in terms of detail]. Figure reffig:afterDataPermute shows the IDG after the update_IDG method for the data permutation transformation has been applied. [Alan: I think that whether we decide to compose two or more data reorderings to the same array should be an ITO].

After data permutation has been applied, the computation specification for the executor in MapIR is as shown in Figure 3. The change in the access relations is due to composing $R_{x \to x'}$ with the current access relations for the `x` and `fx` arrays. Notice that we do not change the name of the arrays. This is so that the statement can have more general formats (e.g. accessing data through macros), and we don't have to recognize these more general constructs and rename them.

4.2 Loop/Computation Alignment

FIXME: We need to look at how the term loop alignment has been used in the past.

FIXME: the below text also describes data alignment a little

Once a loop or data permutation has been performed, it could be that the permuted loop is now accessing a data array indirectly instead of directly and/or other loops that access permuted data arrays are now no longer accessing the

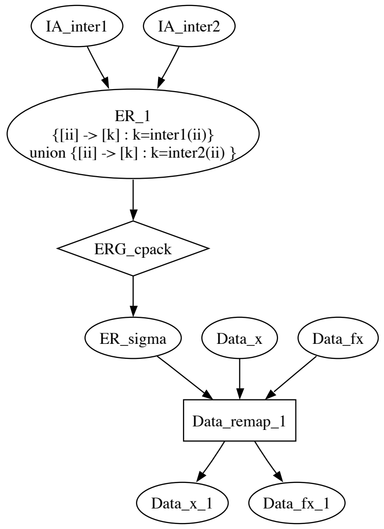

Figure 2: The inspector dependence graph after the compile-time application of data permutation on the data arrays `x` and `fx` based on how `x` is accessed in the `ii` loop.

permuted data array directly. This can be fixed by aligning the data array to the loop that is accessing it or permuting the loop if the loop does not contain any loop carried dependences.

In Figure 3, the status of the access relations after a sequence of transformations are shown. Note that the `i` loop, which originally accessed the `x` and `fx` data arrays direction and sequentially now accesses those data arrays indirectly through the index array `sigma`. Since there are no loop carried dependences in the `i` loop, we can apply an iteration permutation to the `i` loop so as to align the `i` loop with the data arrays `x` and `fx`.

```
moldyn_spec.add_transformation(
    iegen.trans.IterAlignTrans,
    name='iter_align',
    iter_space_trans='''{[c0, s, c0, i, c0] -> [c0, s, c0, j, c0] : c0=0 && j=sigma(i)}
                 union {[c0, s, c1, ii, x] -> [c0, s, c1,ii, x] : c0=0 && c1=1}''')
```

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

$$\begin{aligned} T_{I_0 \to I_1} \quad &= \quad \{[0, s, 0, i, 0] \to [0, s, 0, j, 0] \mid j = sigma(i)\} \\ &\quad\quad \cup \{[0, s, 1, ii, x] \to [0, s, 1, ii, x]\} \end{aligned}$$

Currently the user must specify the transformation for the full iteration space. The transformation relation is needed so as to modify the access relations appropriately.

```
// full iteration space:
//    { [0, s, 0, i, 0 ] : 0<=s && s<T && 0<=i && i<N }
//    union { [0, s, 1, ii, x ] : 0<=s && s<T && 0<=ii
//        && ii<n_inter && 0>=x && x<=1 }
  for (s=0; s<T; s++) {
    for (i=0; i<N; i++) {
S1:     x[sigma[i]] = fx[sigma[i]]*1.25;
    }
    for (ii=0; ii<n_inter; ii++) {
        // simplified computations
S2:     fx[sigma[inter1[ii]]]
            += x[sigma[inter1[ii]]] - x[sigma[inter2[ii]]];
S3:     fx[sigma[inter2[ii]]]
            += x[sigma[inter1[ii]]] - x[sigma[inter2[ii]]];
    }
  }

// data dependences:
//  D_S1_to_S2 =  {[0,s,0,i,0] -> [0,s,1,ii,0] : i = inter1(ii) }
//            union  {[0,s,0,i,0] -> [0,s,1,ii,0] : i = inter2(ii) }
//  D_S1_to_S3 =  {[0,s,0,i,0] -> [0,s,1,ii,1] : i = inter1(ii) }
//            union  {[0,s,0,i,0] -> [0,s,1,ii,1] : i = inter2(ii) }
//  D_S2_to_S1 = { [0,s1,1,ii,0] -> [0,s2,0,i,0] : s2 = s1 + 1
//                && i = inter1(ii) }
//            union  {[0,s1,1,ii,0] -> [0,s2,0,i,0] : s2 = s1 + 1
//                && i = inter2(ii) }
//  D_S3_to_S1 =  { [0,s1,1,ii,1] -> [0,s2,0,i,0] : s2 = s1 + 1
//                && i = inter1(ii) }
//            union  {[0,s1,1,ii,1] -> [0,s2,0,i,0] : s2 = s1 + 1
//                && i = inter2(ii) }
```

Figure 3: Code after the compile-time application of data permutation on the `x` and `fx` data arrays.

Given the parameters for the iteration alignment transformation, the transformation when applied at compile time does the following:

- update_mapIR will not modify the scheduling functions for a loop realignment. Instead the effect of the loop permutation can be seen in the access relations of the loop being permuted. We compose the access relations with the inverse of $T_{I_0 \to I_1}$ to calculate the new access relations. Any data dependences involving iterations in the permuted loop will also be affected by the iteration alignment.
- update_IDG does nothing for loop alignment.

For the example in Figure 3, all of the array accesses in the `i` loop involve the index array sigma, therefore permuting the `i` loop with sigma results in those array accesses no longer needing the indirect access through sigma. Specifically, the access relation for the `x` and `fx` array accesses in loop `i` is as follows:

$$A_{I_0 \to x} = \{[0, s, 0, v, 0] \to [r] \mid r = \mathit{sigma}(v)\}$$

Composing the above with the inverse of $T_{I_0 \to I_1}$ results in the following:

$$A_{I_1 \to x} = A_{I_0 \to x} \text{ compose } (\text{ inverse } T_{I_0 \to I_1})$$

$$\{[0, s, 0, v, 0] \to [r] \mid r = \mathit{sigma}(v)\} \text{ compose } \{[0, s, 0, j, 0] \to [0, s, 0, i, 0] \mid j = \mathit{sigma}(i)\}$$
$$A_{I_1 \to x} = \{[0, s, 0, j, 0] \to [r] \mid r = \mathit{sigma}(v) \wedge j = \mathit{sigma}(v)\}$$

Doing a simplification that sets r equal to j results in a sequential access function.

Iteration reorderings also affect data dependences. To calculate the new data dependences, the iteration reordering, $T_{I_0 \to I_1}$, is composed with the result of the data dependence being composed with the inverse of the same iteration reordering. Assume that $D_{I_0 \to I_0}$ is a dependence relation for the original iteration space I_0. The full set of dependences is the union of `D_S1_to_S2`, `D_S1_to_S3`, `D_S2_to_S1`, and `D_S3_to_S1`. To illustrate how the dependence relations should be modified, we show the effect of the loop alignment transformation on the dependence $D_{I_0 \to I_0} = \{[0, s, 0, i, 0] \to [0, s, 1, ii, 0] \mid i = \mathit{inter}1(ii)\}$, which is a subset of the dependences in the original iteration space.

$$D_{I_1 \to I_1} = T_{I_0 \to I_1} \text{ compose } (D_{I_0 \to I_0} \text{ compose } (\text{ inverse } T_{I_0 \to I_1}))$$

For this example, we have the following (see latex comments for details):

$$\begin{aligned} D_{I_1 \to I_1} \quad = \quad & T_{I_0 \to I_1} \text{ compose} \\ & (\{[0, s, 0, i, 0] \to [0, s, 1, ii, 0] \mid i = \mathit{inter}1(ii)\} \text{ compose} \\ & (\{[0, s, 0, j, 0] \to [0, s, 0, i, 0] \mid j = \mathit{sigma}(i)\} \\ & \cup \{[0, s, 1, ii, x] \to [0, s, 1, ii, x]\})) \end{aligned}$$

which becomes:

$$D_{I_1 \to I_1} = \{[0, s, 0, j, 0] \to [0, s, 1, ii, x] \mid j = \mathit{sigma}(\mathit{inter}1(ii))\}$$

Notice that for iteration alignment, all of the work can be done at compile time. There is no input or output being added to the IDG, because we only end up with the sigma uninterpreted function in the data dependences and access relations. The data permutation transformation already put an explicit relation specification for sigma in the IDG.

FIXME: might want to create a data alignment example to go along with DingKen99.

After loop alignment has been applied at compile time, the inspector dependence graph (IDG) is the same as after data reordering (see Figure 3). After loop alignment has been applied, the computation specification for the executor in MapIR is as shown in Figure 4.

4.3 Iteration permutation

Next we apply an iteration reordering transformation to the `ii` loop. The user provides a description of the transformation as follows:

```
moldyn_spec.add_transformation(
    iegen.trans.IterPermuteTrans,
    name='lexmin',
    iter_space_trans='''{[c0, s, c0, i, c0] -> [c0, s, c0, i, c0] : c0=0 }
          union {[c0, s, c1, ii, x] -> [c0, s, c1, j, x] : c0=0 && c1=1 && j=delta(ii)}
    reordering_name='delta',
    iter_sub_space_relation='{[c0, s, c1, ii, c2] -> [ ii ] : c1=1}',
    target_data_arrays=['x','fx'],
    erg_func_name='ERG_lexmin')
```

The user's description indicates that the `ii` loop in the example should be permuted based on how that loop is accessing the data arrays `x` and `fx`.

The transformation is mathematically specified as a relation on the full iteration space to a new full iteration space as seen here (and as provided by the user):

$$\begin{aligned} T_{I_0 \to I_1} \quad = \quad & \{[0, s, 0, i, 0] \to [0, s, 0, i, 0]\} \\ & \cup \{[0, s, 1, ii, x] \to [0, s, 1, j, x] \mid j = delta(ii)\} \end{aligned}$$

ALAN: the bulleted list below only outlines the algorithm. See the later sub sections for more detail. Given the parameters for the iteration permutation transformation, the transformation when applied at compile time does the following:

- calc_input computes ER_2, which is the access relation that between the loop being permuted and the target arrays.
- The IterationPermuteRTRT.calc_output routine will generate an ER node representing the permutation delta.

```
// full iteration space:
//    { [0, s, 0, i, 0 ] : 0<=s && s<T && 0<=i && i<N }
//    union { [0, s, 1, ii, x ] : 0<=s && s<T && 0<=ii
//        && ii<n_inter && 0>=x && x<=1 }
  for (s=0; s<T; s++) {
    for (i=0; i<N; i++) {
S1:     x[i] = fx[i]*1.25;
    }
    for (ii=0; ii<n_inter; ii++) {
        // simplified computations
S2:     fx[sigma[inter1[ii]]]
            += x[sigma[inter1[ii]]] - x[sigma[inter2[ii]]];
S3:     fx[sigma[inter2[ii]]]
            += x[sigma[inter1[ii]]] - x[sigma[inter2[ii]]];
    }
  }

// data dependences:
//  D_S1_to_S2 =  {[0,s,0,i,0] -> [0,s,1,ii,0] :
//        i = sigma(inter1(ii)) }
//            union  {[0,s,0,i,0] -> [0,s,1,ii,0] :
//        i = sigma(inter2(ii)) }
//  D_S1_to_S3 =  {[0,s,0,i,0] -> [0,s,1,ii,1] :
//        i = sigma(inter1(ii)) }
//            union  {[0,s,0,i,0] -> [0,s,1,ii,1] :
//        i = sigma(inter2(ii)) }
//  D_S2_to_S1 = { [0,s1,1,ii,0] -> [0,s2,0,i,0] : s2 = s1 + 1
//               && i = sigma(inter1(ii)) }
//            union  {[0,s1,1,ii,0] -> [0,s2,0,i,0] : s2 = s1 + 1
//               && i = sigma(inter2(ii)) }
//  D_S3_to_S1 =  { [0,s1,1,ii,1] -> [0,s2,0,i,0] : s2 = s1 + 1
//               && i = sigma(inter1(ii)) }
//            union  {[0,s1,1,ii,1] -> [0,s2,0,i,0] : s2 = s1 + 1
//               && i = sigma(inter2(ii)) }
```

Figure 4: Code after the compile-time application of iteration alignment on the `i` loop. Aligning/permuting the `i` loop to match sigma data reordering.

- update_mapIR will not modify the scheduling functions for a loop permutation, but it will modify all access relations and data dependences affected by the loop permutation.
- update_IDG will add the delta ER and ERG and their dependences into the IDG. The update_IDG method will also iterate over all the relations

in the mapIR and the IDG and ensure that any uninterpreted functions in those relations have a corresponding explicit relation specification in the IDG.

After iteration permutation has been applied, the inspector dependence graph (IDG) is as shown in Figure 5.

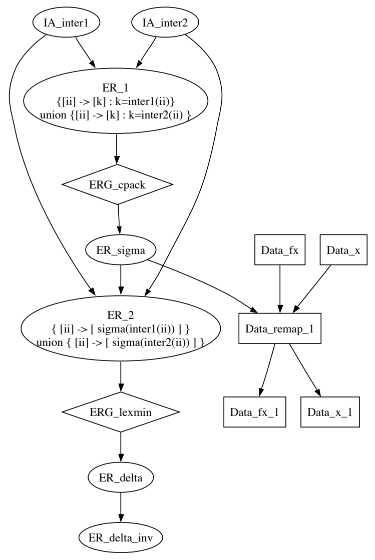
IA_inter1
IA_inter2
ER_1
{[ii] -> [k] : k=inter1(ii)}
union {[ii] -> [k] : k=inter2(ii) }
ERG_cpack
ER_sigma
Data_fx
Data_x
ER_2
{ [ii] -> [sigma(inter1(ii))] }
union { [ii] -> [sigma(inter2(ii))] }
Data_remap_1
ERG_lexmin
Data_fx_1
Data_x_1
ER_delta
ER_delta_inv

Figure 5: The inspector dependence graph after the compile-time application of iteration permutation on the `ii` loop based on how `x` is accessed in the `ii` loop.

After iteration permutation has been applied, the computation specification for the executor in MapIR is as shown in Figure 6.

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 in changes to the access relations and the data dependences.

4.3.1 Calculating the input relation

The iteration permutation RTRT calculates the input access relation (ER_2) by unioning the access relations for all of the statements in the iteration sub space to the target data spaces. To calculate ER_2, first all of the access relations that target the specified target arrays are unioned. For this example, the result of the union over all access relations to `x` and `fx` should be as follows:

$$\begin{aligned} AR &= \{[0,s,0,i,0] \to [i]\} \\ &\quad \cup \{[0,s,1,ii,x] \to [k] \mid k = sigma(inter1(ii))\} \\ &\quad \cup \{[0,s,1,ii,x] \to [k] \mid k = sigma(inter2(ii))\} \end{aligned}$$

```
// full iteration space:
//    { [0, s, 0, i, 0 ] : 0<=s && s<T && 0<=i && i<N }
//    union { [0, s, 1, ii, x ] : 0<=s && s<T && 0<=ii
//            && ii<n_inter && 0>=x && x<=1 }
  for (s=0; s<T; s++) {
    for (i=0; i<N; i++) {
S1:     x[i] = fx[i]*1.25;
    }

    for (ii=0; ii<n_inter; ii++) {
        // simplified computations
S2:     fx[sigma[inter1[delta_inv[ii]]]]
                  += x[sigma[inter1[delta_inv[ii]]]]
                  - x[sigma[inter2[delta_inv[ii]]]];
S3:     fx[sigma[inter2[delta_inv[ii]]]]
                  += x[sigma[inter1[delta_inv[ii]]]]
                  - x[sigma[inter2[delta_inv[ii]]]];
    }
  }

// data dependences:
//  D_S1_to_S2 =  {[0,s,0,i,0] -> [0,s,1,ii,0] :
//        i = sigma(inter1(delta_inv(ii))) }
//            union  {[0,s,0,i,0] -> [0,s,1,ii,0] :
//        i = sigma(inter2(delta_inv(ii))) }
```

Figure 6: Code after the compile-time application of iteration permutation on the ii loop.

Next, the mapping from the full iteration space to the subspace being tiled (iter_sub_space_relation) is used to restrict the domain of the above access relation to the loop being permuted.

$$AR \text{ compose } (\text{ inverse } issr)$$

$$\{[ii] \to [k] \mid k = sigma(inter1(ii))\} \cup \{[ii] \to [k] \mid k = sigma(inter2(ii))\}$$

ALAN: When ER_2 is created, its dependence on sigma, inter1, and inter2 is also noted? Or do we just have a pass that is called in all update_IDG methods? Or a pass that is called after all update_IDG methods? See Post-pass on IDG sub section below for possible answer.

4.3.2 Updating the MapIR

The change in the access relations is due to the following being done to each access relation A originating in the loop being permuted, which for this example is the `ii` loop:

$$A_{I_2 \to x} = A_{I_1 \to x} \text{ compose } (\text{ inverse } T_{I_1 \to I_2})$$

For this example code example, many of the access relations involve the uninterpreted function symbols $sigma$ and $inter1$ or $inter2$. For example,

$$A_{I_1 \to x} = \{[0, s, 1, ii, 0] \to [r] \mid r = sigma(inter1(ii))\}$$

Composition of the access relations with the inverse of the iteration transformations results in relations with an existentially quantified variable as a parameter to two uninterpreted function symbols (see $delta(ii)$ and $inter1(ii)$ in the result of the composition below).

$$\{[0, s, 1, ii, 0] \to [r] \mid r = sigma(inter1(ii))\} \text{ compose } \{[0, s, 1, ii', x] \to [0, s, 1, ii, x] \mid ii' = delta(ii)\}$$

$$A_{I_2 \to x} = \{[0, s, 1, ii', x] \to [r] \mid ii' = delta(ii) \wedge r = sigma(inter1(ii))\}$$

At compile time, the information that the uninterpreted function $delta$ will be a permutation is used to simplify the above equation to the following:

$$\{[0, s, 1, ii', x] \to [r] \mid ii = delta^{-1}(ii') \wedge r = sigma(inter1(ii))\}$$

After the inverse simplification, other simplification rules may be applied to get rid of all existentially quantified variables (e.g. ii).

$$\{[0, s, 1, ii', x] \to [r] \mid r = sigma(inter1(delta^{-1}(ii')))\}$$

Notice that the resulting access relation is reflected in the current version in the code (see Figure 6).

Iteration reorderings also affect data dependences. To calculate the new data dependences, the iteration reordering is composed with the result of the data dependence being composed with the inverse of the iteration reordering.

$$D_{I_2 \to I_2} = T_{I_1 \to I_2} \text{ compose } (D_{I_1 \to I_1} \text{ compose } (\text{ inverse } T_{I_1 \to I_2}))$$

For this example, we have the following (see latex comments for details):

$$\begin{aligned} D_{I_2 \to I_2} \quad = \quad & T_{I_1 \to I_2} \text{ compose} \\ & (\{[0,s,0,i,0] \to [0,s,1,ii,0] \mid i = sigma(inter1(ii))\} \text{ compose} \\ & (\{[0,s,0,i,0] \to [0,s,0,i,0]\} \\ & \cup \{[0,s,1,ii',x] \to [0,s,1,ii,x] \mid ii' = delta(ii)\})) \end{aligned}$$

which becomes:

$$D_{I_2 \to I_2} = \{[0,s,0,i,0] \to [0,s,1,ii',x] \mid ii' = delta(ii) \text{ and } i = sigma(inter1(ii))\}$$

which with the inverse simplification will be simplified to:

$$D_{I_2 \to I_2} = \{[0,s,0,i,0] \to [0,s,1,ii',x] \mid i = sigma(inter1(delta^{-1}(ii')))\}$$

4.3.3 Post-pass on IDG

The inverse simplification introduces `delta_inv` into the updated access relations and data dependences. After the transformation has updated the IDG with any ERG nodes and or inputs and outputs to the ERG nodes, it will still be necessary to determine if any inverse explicit relations are needed in the IDG. For this example, the `delta_inv` ER node will need to be added to the IDG. The IDG will need dependence edge between the `delta` ER node and the `delta_inv` ER node.

(Move later to code gen section: The code generation for the `delta_inv` ER node is a call to the `genInverse` method on the `delta` ER.)

4.4 Sparse Tiling

Next we apply an iteration reordering transformation to both the `i` and `ii` loops. 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. One goal of a sparse tiling transformation is to group iterations such that iterations which reuse the same data are within the same tile and therefore the computation as a whole can experience improved temporal data locality.

For this example, we partition the iterations in the `ii` loop and group iterations in the `i` loop with iterations in the `ii` loop based on those initial partitions. A sparse tiling inspector is responsible for creating an explicit relation, which we will call theta, that maps points in the iteration space to be sparse tiled to tiles. The specific sparse tiling algorithm we use in our experiments is full sparse tiling.

Full sparse tiling is just one way of computing the theta function. Other ways include cache blocking. The overlapping work of Demmel's group is another way, but it places points of computation into two tiles. We can express that with relations, but our code generator does not handle transformations that duplicate iteration points in the target space.

A user can specify a full sparse tiling for the M2 example by adding the sparse tiling transformation to the computation specification.

```
moldyn_spec.add_transformation(
    name = 'blockpart',
    iegen.trans.BlockPart,
    part_name='part',
    num_part='num_tile',    // should become an input symbolic
    iter_sub_space_relation = '{ [c0,s,x,j,y] -> [j] : x=1}',
    erg_func_name='ERG_blockpart')

moldyn_spec.add_transformation(
    name = 'FST',
    iegen.trans.SparseTileTrans,
    tiling_name='theta',
    iter_space_trans='''{[c0, s, c0, i, x] -> [c0, s, c0, t, c0, i, x] : t = theta(0,i)
          union {[c0, s, c1, ii, x] -> [c0, s, c0, t, c1, ii, x] : t = theta(1,ii) && c
    num_tile='num_tile',    // should become an input symbolic
    input_deps_levels = [3],    // level in iteration space before trans for deps that s
    iter_sub_space_relation = '{[c0,s,x,j,y] -> [x,j]}',
    iter_seed_space_relation = '{[c0,s,x,j,y] -> [x,j] : x=1}',
    seed_part = 'part',
    erg_func_name='ERG_fullsparsetile')
```

The above user specification includes the following iteration reordering transformation specification:

$$\begin{array}{rrlcll} T_{I_2\to I_3} & = & \{[0,s,0,i,x] & \to & [0,s,0,t,0,i,x] & \mid t=\theta(0,i)\} \\ & \cup & \{[0,s,1,ii,x] & \to & [0,s,0,t,1,ii,x] & \mid t=\theta(1,ii)\} \end{array}$$

The above user specification also indicates that the block partitioner explicit relation generator (`ERG_blockpart`) should be used to partition the iteration points in the `ii` loop. The symbolic `num_tile` will indicate the number of partitions at runtime. The resulting partitioning will be an explicit relation named `part`.

For the block partition transformation, the following occurs in the calc_input, calc_output, update_IDG and update_mapIR methods of the transformation subclass.

- calc_input: For this example, the partitioner we are going to use in the partitioning step will simply block the iterations of the seed sub space, which is $\{[1,ii] \mid 0 \le ii < numinter\}$. The `calc_output` method associated with `BlockPart` will compute the sub space being partitioned by applying `iter_sub_space_relation` to the full iteration space. The `calc_input` method will then create an ouAny symbolic constants involved in the bounds of the sub space will also be considered input to the partitioning ERG.

- The `calc_output` method will create an explicit relation specification for the partitioning, which maps points in the sub space to be partitioned to partitions.
- Calling `update_IDG` on the block partitioning subclass will cause insertion of the nodes ERG_blockpart and ER_part and their associated dependences into the IDG.
- The block partitioning transformation does not modify the computation specification in any way.

The full sparse tiling algorithm (ERG_fullsparsetile) will use the partitioning stored in `part` as its seed partition and will inspect all dependences to and from the seed space within the sub space of the full computation that is being sparse tiled. For now, the dependences that are input to the sparse tiling algorithm are specified by having the user indicate at which level the dependences are carried. In this example, the dependences between loops `i` and `ii` are the dependences that will be inspected.

The following is a description of how the sparse tiling transformation will add nodes and edges to the IDG and modify the MapIR.

- The sparse tiling transformation's `calc_input` method creates an explicit relation specification node for the dependences within the sub space being sparse tiled that target the partitioning subspace and that originate from the partitioning subspace.
- The sparse tiling transformation `calc_output` routine will generate an ER node representing the tiling function θ.
- The `update_IDG` method inserts an ERG node into the IDG to indicate a call to a specific sparse tiling algorithm. Dependences between the ERG and its inputs and outputs are also inserted.
- The sparse tiling transformation `update_mapIR` will apply the transformation specification $T_{I_1 \to I_2}$ to the statement scheduling functions and access functions.

After the sparse tiling has been applied at compile time, the inspector dependence graph (IDG) is as is shown in Figure 9. After sparse tiling has been applied, the computation specification for the executor in MapIR is as shown in Figure 10.

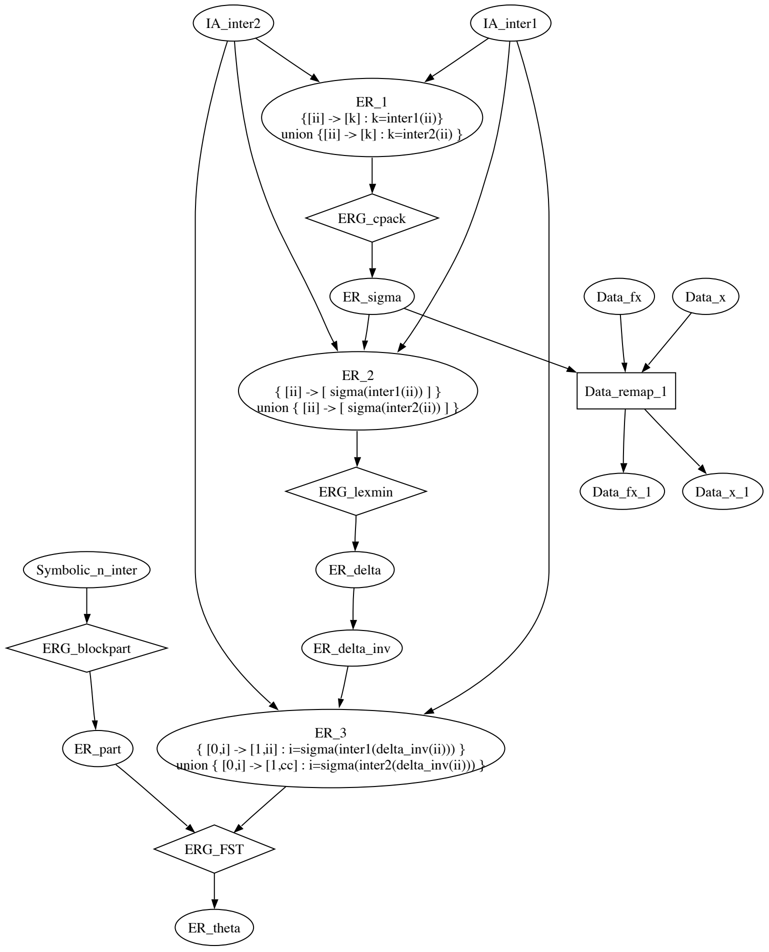
IA_inter2
IA_inter1
ER_1
{[ii] -> [k] : k=inter1(ii)}
union {[ii] -> [k] : k=inter2(ii) }
ERG_cpack
ER_sigma
Data_fx
Data_x
ER_2
{ [ii] -> [sigma(inter1(ii))] }
union { [ii] -> [sigma(inter2(ii))] }
Data_remap_1
ERG_lexmin
Data_fx_1
Data_x_1
Symbolic_n_inter
ER_delta
ERG_blockpart
ER_delta_inv
ER_3
{ [0,i] -> [1,ii] : i=sigma(inter1(delta_inv(ii))) }
union { [0,i] -> [1,cc] : i=sigma(inter2(delta_inv(ii))) }
ER_part
ERG_FST
ER_theta

Figure 7: The inspector dependence graph after the compile-time application of sparse tiling on the `i` and `ii` loops based on the dependences between the two points.

4.4.1 Calculating the inputs to sparse tiling

When the user specifies the data dependence relations, he or she will also specify at what level in the full iteration space the data dependence is carried. For example, the dependence between statements 1 and 2 after the iteration permutation of the `ii` loop is as follows:

$$\text{D\_S1\_to\_S2} \;= \{[0,s,0,i,0] \rightarrow [0,s,1,ii',x] \mid i = sigma(inter1(delta^{-1}(ii')))\}$$

The above dependence is carried by the third element in the full iteration space vector, because the dependence is between the `i` and `ii` loops and is not carried by the `s` loop.

```
// full iteration space:
//    { [0, s, 0, t, 0, i, 0 ] : 0<=s && s<T && 0<=i && i<N
                                   && t=theta(0,i) && 0<=t<=nt }
//    union { [0, s, 0, t, 1, ii, x ] : 0<=s && s<T && 0<=ii && ii<n_inter && 0>=x && x<=1
                                   && t=theta(1,ii) && 0<=t<nt }
  for (s=0; s<T; s++) {
    for (t=0; t<nt; t++) {
      for (i=0; i<N; i++) {
S1:     if (t=theta(0,i)) { x[i] = fx[i]*1.25; }
      }

      for (ii=0; ii<n_inter; ii++) {
        // simplified computations
S2:     if (t=theta(1,ii)) { fx[sigma[inter1[delta_inv[ii]]]]
                  += x[sigma[inter1[delta_inv[ii]]]]
                  - x[sigma[inter2[delta_inv[ii]]]]; }
S3:     if (t=theta(1,ii)) { fx[sigma[inter2[delta_inv[ii]]]]
                  += x[sigma[inter1[delta_inv[ii]]]]
                  - x[sigma[inter2[delta_inv[ii]]]]; }
      }
  }

// data dependences:
//  D_S1_to_S2 =  {[0,s,0,t,0,i,0] -> [0,s,0,t,1,ii,0] :
//       i = sigma(inter1(delta_inv(ii))) }
//           union  {[0,s,0,t,0,i,0] -> [0,s,0,t,1,ii,0] :
//       i = sigma(inter2(delta_inv(ii))) }
```

Figure 8: Code after the compile-time application of iteration permutation on the ii loop.

The sparse tiling transformation specification indicates that the inspector should only inspect data dependences carried by the third element in the full iteration space. Thus we compute the relation to inspect by unioning the above dependence and the dependences between statement 1 and 3, but not the dependences between statements 2 and 1, and the dependences between statements 3 and 1 since those are carried by the `s` loop.

The union of the relevant dependences is as follows:

$$\begin{aligned} D \quad = \quad & \{[0,s,0,i,0] \rightarrow [0,s,1,ii,0] \mid i = sigma(inter1(delta^{-1}(ii)))\} \\ & \text{union } \{[0,s,0,i,0] \rightarrow [0,s,1,ii,0] \mid i = sigma(inter2(delta^{-1}(ii)))\} \end{aligned}$$

See doc/sparse-tile-design.txt for more details about how the `FROM_SEED` and `TO_SEED` dependences are computed.

4.4.2 Updating the MapIR

The sparse tiling transformation causes the schedules for the statements in the sub space being sparse tiled to be modified. For example, S1 starts with the following schedule/scattering function

$$\{[s,i]- > [0,s,0,i,0]\}.$$

The full sparse tiling transformation should be composed with this original scheduling function to create the updated schedule for S1,

$$T_{I_2 \rightarrow I_3} \text{ compose } \{[s,i] \rightarrow [0,s,0,i,0]\}$$

which equals

$$\{[s,i] \rightarrow [0,s,0,t,0,i,0] : t = theta(0,i)\}$$

The access relations do not appear to change in the code because the array references do not change. However, the access relations need to be modified so that they are mapping the new full iteration space to the various arrays being accessed. The access relations are modified by the iteration reordering transformation due to sparse tiling in the same way they were modified for iteration alignment and iteration permutation. For this example, new set of access relations from the third version of the full iteration space to the data array `x` are computed as follows:

$$A_{I_3 \rightarrow x} = A_{I_2 \rightarrow x} \text{ compose (inverse } T_{I_2 \rightarrow I_3})$$

which results in the following:

$$A_{I_3 \rightarrow x} =$$

FIXME: we end up with t=theta(...) constraints in the access relations when we shouldn't really need these because of the fact that the iteration space will have that constraint.

The affect on the data dependences is computed the same as was done after iteration alignment and iteration permutation; the iteration reordering is composed with the result of the data dependence being composed with the inverse of the iteration reordering.

$$D_{I_3 \to I_3} = T_{I_2 \to I_3} \text{ compose } (D_{I_2 \to I_2} \text{ compose } (\text{ inverse } T_{I_2 \to I_3}))$$

For this example, we have the following (see latex comments for details): FIXME

$$\begin{aligned} D_{I_2 \to I_2} \quad = \quad & T_{I_1 \to I_2} \text{ compose} \\ & (\{[0,s,0,i,0] \to [0,s,1,ii,0] \mid i = sigma(inter1(ii))\} \text{ compose} \\ & (\{[0,s,0,i,0] \to [0,s,0,i,0]\} \\ & \cup \{[0,s,1,ii',x] \to [0,s,1,ii,x] \mid ii' = delta(ii)\})) \end{aligned}$$

which becomes:

$$D_{I_2 \to I_2} = \{[0,s,0,i,0] \to [0,s,1,ii',x] \mid ii' = delta(ii) \text{ and } i = sigma(inter1(ii))\}$$

which with the inverse simplification will be simplified to:

$$D_{I_2 \to I_2} = \{[0,s,0,i,0] \to [0,s,1,ii',x] \mid i = sigma(inter1(delta^{-1}(ii')))\}$$

4.5 Inter Transformational Optimizations (ITO)

Requirements on the inspector dependence graph.

- Each explicit relation specification node should have an input arc from another explicit relation or index array node for each uninterpreted function symbol in the specification. For example, ER_2 requires IA_inter2, IA_inter1, and ER_sigma as input.
- If an explicit relation specification is modified, it must be equivalent?

Most critical ones (we need data to back this list up)

- Index array collapsing.
- Sparse loops.

Others

- Remap data only once. When a string of two or more data remappings appear in the inspector dependence graph, then the remap once ITO will create an explicit relation specification for the composition of all the relevant data permutations and make is so that the data remapping only occurs once.

4.5.1 Index Array Collapsing, or Pointer Update

Index array collapsing is just a matter of inserting an ER specification for nested ERS and replacing those nested uninterpreted function symbols in other ER specifications (including those in the MapIR) and putting the appropriate edges in the IDG. Index array collapsing is strictly more general than pointer update [?], which focused on modifying index arrays after a data permutation so as to reduce the level of indirection in a loop to one [FIXME: verify this]. Index array collapsing applies any time there are two or more nested uninterpreted function symbols in any explicit relation specifications. Such nesting occurs more generally, for example ...

In the example, the explicit relation specification that is input into the lexmin iteration permutation algorithm involves the nested uninterpreted function symbols $sigma(inter1(ii))$ and $sigma(inter2(ii))$.

Thought experiment: What if we just treat each ITO as just another transformation?

```
moldyn_spec.add_transformation(
    iegen.trans.IndexArrayCollapse,
    name='sigmainter1',
    outer_func='sigma',
    inner_func='inter1',
    erg_func_name='ERG_compose')  // not sure if we need this to be explicitly named.  N
```

Given the parameters for the index array collapsing transformation, the transformation when applied at compile time does the following:

- update_IDG will create an explicit relation specification (see ER_4 in Figure ??)

  access relation (ER_2) by unioning the access relations for all of the statements in the iteration sub space to the target data spaces. Note that for this example the access relations were modified due to the previously applied data permutation. When ER_2 is created, its dependence on sigma, inter1, and inter2 is also noted.

- The IterationPermuteRTRT.calc_output routine will generate an ER node representing the permutation delta.

- update_mapIR will not modify the scheduling functions for a loop permutation. This is because we still want the loop bounds to stay affine(?). Instead the effect of the loop permutation can be seen in the access relations of the loop being permuted. We compose the access relations with the inverse of $T_{I_0 \to I_1}$ to calculate the new access relations. Simplification steps result in an inverse of *delta* being computed. Any data dependences involving iterations in the permuted loop will also be affected by the iteration reordering.

- update_IDG will add the ER and ERG and their dependences into the IDG. Will also need node for the inverse simplification.

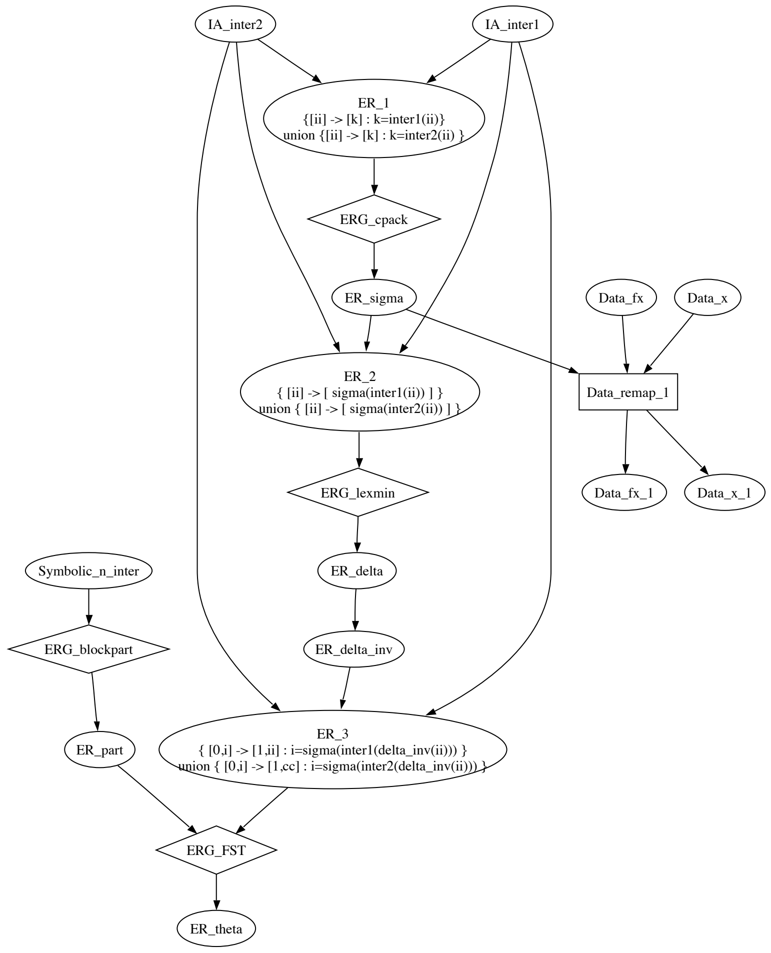
IA_inter2
IA_inter1
ER_1
{[ii] -> [k] : k=inter1(ii)}
union {[ii] -> [k] : k=inter2(ii) }
ERG_cpack
ER_sigma
Data_fx
Data_x
Data_remap_1
ER_2
{ [ii] -> [sigma(inter1(ii))] }
union { [ii] -> [sigma(inter2(ii))] }
ERG_lexmin
Data_fx_1
Data_x_1
Symbolic_n_inter
ER_delta
ERG_blockpart
ER_delta_inv
ER_part
ER_3
{ [0,i] -> [1,ii] : i=sigma(inter1(delta_inv(ii))) }
union { [0,i] -> [1,cc] : i=sigma(inter2(delta_inv(ii))) }
ERG_FST
ER_theta

Figure 9: The inspector dependence graph after the compile-time application of sparse tiling on the `i` and `ii` loops based on the dependences between the two points.

4.5.2 Discussion

Order and selection matters and is an open problem. I am guessing that if we do the index array collapsing before we do the data and computation alignment, we will end up with more work in the inspector. Also if we apply loop realignment after tiling, then we have to modify the tiling function theta.

4.6 Generating the composed inspector

Code generation of the composed inspector.

```
// full iteration space:
//    { [0, s, 0, t, 0, i, 0 ] : 0<=s && s<T && 0<=i && i<N
                                    && t=theta(0,i) && 0<=t<=nt }
//    union { [0, s, 0, t, 1, ii, x ] : 0<=s && s<T && 0<=ii && ii<n_inter && 0>=x && x<=1
                                    && t=theta(1,ii) && 0<=t<nt }
  for (s=0; s<T; s++) {
    for (t=0; t<nt; t++) {
      for (i=0; i<N; i++) {
S1:     if (t=theta(0,i)) { x[i] = fx[i]*1.25; }
      }

      for (ii=0; ii<n_inter; ii++) {
        // simplified computations
S2:     if (t=theta(1,ii)) { fx[sigma[inter1[delta_inv[ii]]]]
                  += x[sigma[inter1[delta_inv[ii]]]]
                  - x[sigma[inter2[delta_inv[ii]]]]; }
S3:     if (t=theta(1,ii)) { fx[sigma[inter2[delta_inv[ii]]]]
                  += x[sigma[inter1[delta_inv[ii]]]]
                  - x[sigma[inter2[delta_inv[ii]]]]; }
      }
  }
```

Figure 10: Code after the compile-time application of iteration permutation on the `ii` loop.

4.7 Generating the composed executor

Code generation of the composed executor.