Compile-Time Composition of Run-time Data and Iteration Reorderings

Michelle Mills Strout
UC, San Diego
CSE Dept, 9500 Gilman Dr.
La Jolla, CA 92037-0114
mstrout@cs.ucsd.edu

Larry Carter
UC, San Diego
CSE Dept, 9500 Gilman Dr.
La Jolla, CA 92037-0114
carter@cs.ucsd.edu

Jeanne Ferrante
UC, San Diego
CSE Dept, 9500 Gilman Dr.
La Jolla, CA 92037-0114
ferrante@cs.ucsd.edu

ABSTRACT

Many important applications, such as those with indirect memory references or sparse data structures, have memory reference patterns which are unknown at compile-time. To exploit locality in such applications, prior work has developed run-time reorderings to transform the computation and data.

This paper presents a compile-time framework that allows the explicit composition of run-time data and iteration reordering techniques for locality. Our framework builds on the iteration reordering framework of Kelly and Pugh to represent the effects of a given composition. Using this representation, data need be remapped *only once* at runtime for a given composition. Since sparse tiling techniques (developed by us and others) are included in our framework, they become more generally applicable, both to a larger class of applications, and in their composition with other reordering transformations. We show that new compositions suggested by our framework can result in better overall performance than previous work on three application benchmarks. For instance, composing intra-loop transformations with sparse tiling techniques can improve inter-loop locality. Our experiments show that significant performance improvements can result, with little increase in overhead.

1. INTRODUCTION

Data locality and parallelism are essential for improving the performance of applications on current architectures. Data and loop transformations can fu rther both goals. Until recently the focus has been primarily on compile-time transformation frameworks [15, 24, 14, 4, 12, 13, 29, 11, 28] restricted to affine loop bounds and affine array references.

One such framework is that of Kelly and Pugh [12], which represents loop nests as iteration spaces. A compiler can use their framework to transform iteration spaces and implement corresponding data reorderings. The legality of such transformations is determined by the data dependences of the program. Previous compile-time frameworks (such as theirs) do not handle non-affine memory references. Nevertheless, such memory references occur in many important applications, such as sparse matrix computations and unstructured mesh applications [18]. Fortunately, the Kelly and Pugh framework allows the description of indirect memory references (such as `A[B[i]]`) through the use of Presburger arithmetic with uninterpreted function symbols [20]. We exploit this ability to specify *data mappings* and *dependences* between interations involving indirect memory references.

Describing the effect of run-time iteration and data reorderings in a compile-time framework provides several advantages. First, both run-time and compile-time transformations can be described in the same framework. Secondly, it may be possible to reduce the number of times the data is moved to new locations at runtime, even when using several reordering transformations.

We build on the Kelly and Pugh framework to describe the effects of run-time data and iteration reordering transformations for locality, which include consecutive packing [6], graph partitioning [9], bucket-tiling [18], lexicographical grouping [6], full sparse tiling [26], and cache blocking [7]. We show how the symbolic effect of a run-time transformation can be propagated to relevant data mappings and dependences. Given a selected composition of run-time reorderings, the resulting data mappings and dependences can be used by any subsequent choice of run-time transformation. While this paper focuses on data locality, our framework can also be used to describe run-time transformations for parallelism as well.

Formalizing run-time iteration and data reorderings is only one step towards our goal of automating the creation of composite runtime transformations for sparse computations. Still needed are methods of automatically generating runtime inspectors for a variety of source

code idioms. Another key component is guidance mechanisms that decide when to apply which sequence of transformations. Nevertheless, we believe that the framework presented here helps make such a system possible, and our experiments (on the IRREG, NBF, and MOLDYN benchmarks [9]) illustrate that, in some cases, large performance improvements can be made.

Summarizing, this paper makes the following contributions:

- We show how to use an existing compile-time framework to describe a number of run-time iteration and data reordering transformations that focus on intra-loop locality. We also show how this approach leads to new combinations of optimizations.
- We show that sparse tiling techniques, which improve inter-loop locality, can be described in this framework. As a result, sparse tiling can be applied to a larger class of programs (until now, it has only been applied to Gauss-Seidel).
- We give experimental results (using hand-coded versions which we believe can ultimately be automatically generated) that show that significant performance improvements can result from the combined transformations.

Section 2 reviews the terminology for the Kelly and Pugh iteration reordering framework using an example irregular kernel (used throughout the paper). Section 3 illustrates iteration and data reordering transformations on a single loop. We present new combinations of these run-time transformations that can result in less overhead and improved performance. In section 4 we show how sparse tiling in combination with other run-time transformations can be applied to multiple outer loop iterations, and can result in even better performance with little increase in overhead. Section 6 describes related work. Finally, in section 7 we discuss future work and conclude.

2. FRAMEWORK TERMINOLOGY

In this section, we review the Kelly-Pugh iteration reordering framework terminology using an example irregular computation. Figure 1 shows a simplified version of the Moldyn kernel, which is a molecular dynamics code. The loop containing statement $S1$ makes a copy of the data array `y` into the data array `y`. Then the `j` loop visits the `left` and `right` arrays, which give pairs of molecules that are close enough to interact. Typically the number of interactions is a small percentage of the total possible number of interactions. Previous work [6, 9] refers to `left` and `right` as the access or *index* arrays, and `x` and `y` as base or *data* arrays.

2.1 Loops, Statements, and Data

The traditional literature on loop transformations represents each iteration within a loop nest as an integer tuple, $[i_1, ..., i_n]$, where i_p is the value of the iteration variable for the pth loop. Thus, it is a set of integer tuples with constraints indicating the loop bounds.

$$\{[i_1, ..., i_n] | lb_1 \leq i_1 \leq ub_1 \wedge \cdots \wedge lb_n \leq i_n \leq ub_n\}$$

```
do t = 1 to num_steps
   do i=1 to num_nodes
S1    x[i] = y[i]
   enddo

   do j=1 to num_inter
S2    y[left[j]] += f(x[left[j]], x[right[j]])
S3    y[right[j]] += f(x[left[j]], x[right[j]])
   enddo
enddo
```

Figure 1: Simplified Moldyn Example

This representation is not particularly convenient for representing transformations that operate on a collection of loops that are not perfectly nested. For instance, there are two traditional iteration spaces in the code shown in figure 1. Ahmed et. al. [1] and Kelly-Pugh [12] suggest two different methods for constructing a unified iteration space. In this paper we illustrate the our methods using the Kelly-Pugh method. For the simple Moldyn example, they would use a four-dimensional space. Each loop corresponds to a pair of dimensions, where the first dimension of the pair is a value of the index variable, and the second is a number representing the statement within the loop. A program executes its iterations in lexicographic order of the unified iteration space.

For instance, using this representation, the $[t, j]$-th iteration of S2 is denoted $[t, 2, j, 1]$ since S2 is in the second statement of the outer loop, and it's the first statement of the inner loop. The unified iteration space I_0 for the (untransformed) program is the following set:

$$\begin{aligned} I_0 = \quad & \{[t,1,i,1] \mid \quad 1 \leq t \leq num\_steps \\ & \qquad\qquad \wedge 1 \leq i \leq num\_nodes\} \cup \\ & \{[t,2,j,q] \mid \quad 1 \leq t \leq num\_steps \\ & \qquad\qquad \wedge 1 \leq j \leq num\_inter \wedge 1 \leq q \leq 2\} \end{aligned}$$

Next we will define data mappings and dependences for the unified iteration space.

2.2 Data Mappings

Each array has an associated data space represented with a integer tuple set with the same dimensionality as the array. The simplified Moldyn example contains 4 data spaces:

$$\begin{aligned} x_0 &= \{[i] \mid 1 \leq i \leq num\_nodes\} \\ y_0 &= \{[i] \mid 1 \leq i \leq num\_nodes\} \\ left_0 &= \{[i] \mid 1 \leq i \leq num\_inter\} \\ right_0 &= \{[i] \mid 1 \leq i \leq num\_inter\} \end{aligned}$$

The subscripts "0" are used here since these are the arrays used in the original, untransformed program.

Define a *data mapping* $M_{I_0 \to a}$ from iterations to sets of storage locations in an array a, so that for each iteration $p \in I_0$, $M_{I_0 \to a}(p)$ is the set of locations that are referenced by iteration p. Notice that the subscript "$I_0 \to a$" gives the domain and range of the mapping.

The Moldyn example has the following data mappings:

$$\begin{aligned} M_{I_0 \to x_0} &= \{[t,1,i,1] \to [i]\} \\ &\quad \cup \{[t,2,j,q] \to [left(j)]\} \\ &\quad \cup \{[t,2,j,q] \to [right(j)]\} \\ M_{I_0 \to y_0} &= \{[t,1,i,1] \to [i]\} \\ &\quad \cup \{[t,2,j,1] \to [left(j)]\} \\ &\quad \cup \{[t,2,j,2] \to [right(j)]\} \\ M_{I_0 \to left_0} &= \{[t,2,j,q] \to [j]\} \\ M_{I_0 \to right_0} &= M_{I_0 \to left_0} \end{aligned}$$

Figure 2 uses circles to represent the iterations of statements `S1` and `S2`. The dotted lines indicate some of the data mapping relationships.

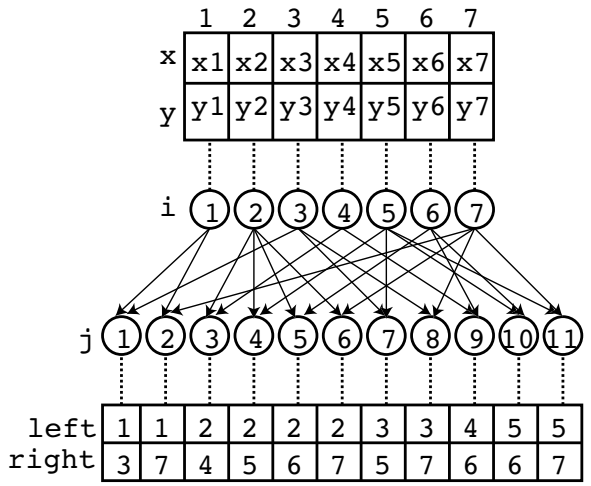

Figure 2: Iteration space for `i` and `j` loops of Moldyn example. Here, circles represent iterations of statements S1 and S2 inside one iteration of the outer loop. Arrows represent dependences between the iterations, and dotted lines represent the data mapping.

2.3 Dependences

Define the *dependences* D_I to be the set of directed edges between iterations that represent dependent computations. For example, the dependences between statements `S1` ($[t,1,i,2]$) and `S2` ($[t,2,j,1]$) due to the `x` and `y` arrays are specified with the following dependence relation.

$$\begin{aligned} D_{I_0 \to I_0} = \{[t,1,i,1] \to [t',2,j,2] \mid t \le t' \\ \wedge (i = left(j) \ \vee \ i = right(j))\} \end{aligned}$$

The arrows in figure 2 represent these dependences.

3. MANIPULATING AND COMBINING RUN-TIME TRANSFORMATIONS

Many run-time transformations make use of *inspectors*. An inspector is code that examines run-time information (often some of the program's index arrays) to produce a *reordering*, which will later be applied to data mappings (in the case of a *data reordering* transformation) or to the order that iterations will be executed (for an *iteration reordering*.) Figure 3, described later, is an example of an inspector.

With compile-time transformations, it is only necessary to manipulate the mapping from program statements to a unified iteration space to show the composition of various compile-time transformations. With run-time transformations, an inspector may take as input a reordering that was produced by an earlier inspector. Thus, we want our framework to describe the data mappings and dependences in effect at all stages of the transformation process. However, it may not be necessary to implement every data reordering at runtime. For instance, rather than reordering an array several times, it may be possible to compose the reorderings, then apply a *remapper* at the end. This can be a substantial savings, particularly when each element of a data array is a large structure.

It is up to the compiler to judiciously choose when to apply a remapper to a data array, and when it can be postponed or avoided. Our framework lets the compiler explore these possibilities by expressing the effects of each run-time reordering transformation as a mapping from an iteration space (either given by the original execution order or after some transformations have been applied) to another unified iteration space.

Formally, an *iteration reordering* transformation is a mapping $T_{I \to I'}$ that assigns each iteration p of iteration space I to iteration $T_{I \to I'}(p)$ in a new iteration space I'. The new execution order is given by the lexicographic order of the iterations in I'.

When we consider a iteration-reordering transformation, we must verify that the new execution order respects all the dependences of the original. Thus for each $\{p_1 \to p_2\} \in D_{I \to I}$, $T_{I \to I'}(p_1)$ must be lexicographically earlier than $T_{I \to I'}(p_2)$.

The dependences $D_{I' \to I'}$ of the new iteration space are given by

$$D_{I' \to I'} = \{T_{I \to I'}(p_1) \to T_{I \to I'}(p_2) \mid p_1 \to p_2 \in D_{I \to I}\}$$

and the new data mapping $M_{I' \to a}$ for each array a is

given by

$$M_{I' \to a} = \{T_{I \to I'}(p) \to M_{I \to a}(p) \mid p \in I\}$$

Having constructed the new dependences and data mappings, we are ready to plan further run-time transformations.

A *data reordering transformation* corresponds to another mapping $R_{a \to a'}$, where the data that was originally stored in location m is relocated to $R_{a \to a'}(m)$. We do not need to consider the legality of a data mapping - any one-to-one data remapping is legal. A corollary is that the data dependences do not change after a data reordering.

After remapping an array a, new data mapping $M_{I \to a'}$ is given by

$$M_{I \to a'} = \{q \to R(m) | m \in M_{I \to a}(q)\}$$

The next sections illustrate how to specify the effects of applying several intra-loop run-time data and iteration reordering transformations for the simplified Moldyn example. Figure 5 give pseudo-code for the resulting composed inspector, which will be referred to throughout the section.

3.1 Run-Time Data Reordering

Given a loop with indirect memory references, run-time data reordering techniques attempt to improve the spatial and temporal data locality in the loop by reordering the data based on the order in which it is referenced in the loop. Consecutive packing (CPACK [6]) and graph partitioning (Gpart [9]) are two example data reordering transformations discussed in this paper. Other run-time data reordering transformations include Reverse Cuthill-McKee [5, 2] and space-filling curves [25, 17].

A CPACK inspector traverses the data mappings for the loop with indirect memory references in lexicographical order of the loop iteration space. The first time the loop touches a piece of data, that data is packed into the next location for the new data mapping. Figure 3 shows the CPACK inspector code specialized for the original data mapping $M_{I_0 \to x_0}$ (specified in section 2.2) for the simplified Moldyn example. This specialized CPACK inspector is called by the composed inspector in figure 5.

Iteration run-time reordering inspectors for intra-loop locality can also be implemented by traversing these data mappings. This key insight allows us to describe many run-time data and iteration transformations within the same framework, and is the first step towards automatic generation of specialized inspectors.

The effect of CPACK can be specified at compile-time by changing all the data mappings which involve the array being reordered. In the simplified moldyn example, it makes sense to construct the same reordering for the `x` and `y` arrays. Let $R_{x_0 \to x_1} = \{i \to \sigma_{cp}(i)\}$ specify the run-time data reordering on the arrays `x` and `y`,

```
CPACK_M_IO_to_x0(left,right)
  // initialize alreadyOrdered bit
  // vector to all zeros
  do j=1 to num_inter
  mem_loc1 = left[j]
  mem_loc2 = right[j]

  if !alreadyOrdered(mem_loc1)
     sigma_cp_inv[count] = mem_loc1
     alreadyOrdered(mem_loc1) = true
     count = count + 1
  endif

  if !alreadyOrdered(mem_loc2)
     sigma_cp_inv[count] = mem_loc2
     alreadyOrdered(mem_loc2) = true
     count = count + 1
  endif
  enddo
  return sigma_cp_inv
```

Figure 3: First CPACK inspector for moldyn

where x_0 is the `x` array in its original order, and x_1 is the reordered `x` array. The new data mapping, $M_{I \to x_1}$, is specified as follows.

$$\begin{aligned} M_{I_0 \to x_0} \quad = \quad & \{[t,1,i,1] \to [\sigma_{cp}(i)]\} \\ & \cup \{[t,2,j,q] \to [\sigma_{cp}(left(j))]\} \\ & \cup \{[t,2,j,q] \to [\sigma_{cp}(right(j))]\} \end{aligned}$$

Partitioning algorithms like Gpart [9] logically operate on a graph where each data location is a node. There is an edge between two nodes whenever their associated data is touched within the loop with indirect memory references. Gpart is a heuristic partitioning algorithms which attempts to make partitions of the graph. The goal is to make the data associated with one partition fit into (some level of) cache.

A data reordering based on Gpart σ_{gp} orders data within the same partition consecutively. In the simple moldyn example, a Gpart data reordering will change the iteration to data space mappings and data dependences in the same way CPACK did, just replace σ_{cp} with σ_{gp}.

3.2 Run-Time Iteration Reordering

Typically a iteration reordering of a loop with indirect memory accesses follows a data reordering like CPACK. Both techniques improve the locality within the loop – intra-loop locality.

The goal of iteration reordering is to reorder the loop which contains the indirect memory references based on the order in which the loop will touch the data.

One such iteration reordering is lexicographical grouping (lexGroup) [6]. In lexGroup all of the computations

which index into one data location are executed before moving on to the computations touching the next data locations. The lexGroup inspector takes into account any previously created data reordering. Therefore, lexGroup will iterate over the data mappings which include the σ_{cp} function if lexGroup is performed after CPACK.

The iteration reordering of the `i` loop and the `j` loop based on their mappings to the data arrays `x` and `y` is specified as follows.

$$\begin{aligned} T_{I_0 \to I_1} &= \{[t,1,i,1] \to [t,1,\sigma_{cp}(i),1]\} \\ &\quad \cup [t,2,j,k] \to [t,2,\sigma_{lg}(j),k]\} \end{aligned}$$

Notice that reordering the `i` loop does not require any extra code in the composed inspector (figure 5). Since each iteration of the `i` loop directly maps to the `x` and `y` arrays, the reordering function generated for them, σ_{cp}, can also be used to reorder the `i` loop.

This reordering is legal because there are no dependences in either loop which disallow reordering of the loop. Only reduction dependences occur between iterations of the `j` loop.

The data mappings for our example code change again, due to the iteration reordering.

$$\begin{aligned} M_{I_1 \to x'} &= \{[t,1,i,1] \to [\sigma_{cp}(i)]\} \\ &\quad \cup \{[t,2,\sigma_{lg}(j),q] \to [\sigma_{cp}(left(j))]\} \\ &\quad \cup \{[t,2,\sigma_{lg}(j),q] \to [\sigma_{cp}(right(j))]\} \end{aligned}$$

The data dependences change as well. Below we show the effect of reordering the `j` loop on the data dependences between `S1` and `S2`.

$$\begin{aligned} D_{I_1 \to I_1} = \{[t,1,i,1] \to [t',2,\sigma_{lg}(j),2] \mid t \leq t' \\ \wedge (i = left(j) \ \vee \ i = right(j))\} \end{aligned}$$

3.3 Benefits of compile-time descriptions

With a compile-time description of the effects of a run-time data or iteration reordering, it is possible to plan compositions of run-time transformations and generate code for the composed data remappings at the end of all inspection.

When combining run-time transformations, specific instances of the relevant inspectors can be created to take into account changes to the data space mappings and dependences incurred by the previously planned inspector. The explicit description of how various run-time mappings affect each other suggests different combinations which have not been tried before. For example, it is possible to execute another CPACK data reordering after executing CPACK and lexGroup.

Figure 4 shows how the second CPACK inspector is specialized to traverse the current data mappings $M_{I_1 \to x_1}$. The second CPACK inspector will specify a data reordering $R_{x_0 \to x_2} = \{i \to \sigma_{cp2}(i)\}$, where x_0 is the `x` array in its original order.

```
CPACK_M_I1_to_x1(left,right,sigma_cp,
        sigma_lg_inv)
  // initialize alreadyOrdered bit
  // vector to all zeros
  do k=1 to num_inter
  mem_loc1 = sigma_cp[left[sigma_lg_inv[k]]]
  mem_loc2 = sigma_cp[right[sigma_lg_inv[k]]]

  if !alreadyOrdered(mem_loc1)
     sigma_cp2_inv[count] = mem_loc1
     alreadyOrdered(mem_loc1) = true
     count = count + 1
  endif

  if !alreadyOrdered(mem_loc2)
     sigma_cp2_inv[count] = mem_loc2
     alreadyOrdered(mem_loc2) = true
     count = count + 1
  endif
  enddo
  return sigma_cp2_inv
```

Figure 4: Second CPACK inspector for moldyn

Manipulating mapping arrays (`sigma_cp`, `sigma_lg`, etc.) at run-time allows us to explicitly wait until all inspection is complete to reorder the data. The composed inspector in figure 5 reorders the data and index arrays accordingly after all data and iteration reorderings have been computed. Not shown in this example, but implemented in the experiments, it is possible to do another data reordering after the sparse tiling inspector.

4. GENERALIZED SPARSE TILING

Sparse tiling techniques, full sparse tiling [26] and cache blocking [7], were developed for an important kernel used in Finite Element Methods, Gauss-Seidel. Sparse tiling results in run-time generated tiles or iteration slices [21] which cut across loops that only touch a subset of the total data. By performing a iteration reordering based on the sparse tiling, inter-loop locality can improve. Intuitively, sparse tiling is applicable anytime a pair of loops share an outer loop (in order to amortize the overhead of run-time iteration reordering) and there are dependences between the the inner loops.

Sparse tiling differs from the previously discussed iteration reordering transformations in four ways.

- Sparse tiling improves the inter-loop data locality whereas the other techniques discussed in this paper only affect the data locality within individual loops, intra-loop data locality.
- Until now, sparse tiling techniques have only been applied to Gauss-Seidel. By specifying the effect of sparse tiling within our composition framework, the legality of applying sparse tiling in any code is possible.

```
// Specialized Inspector Sequence

// First application of CPACK
sigma_cp_inv = CPACK_M_I0_to_x0(left0,
                right0)
sigma_cp     = calcInverse(sigma_cp_inv)

// Application of lexGroup
sigma_lg     = lexGroup_M_I0_to_x1(left0,
                right0,sigma_cp)
sigma_lg_inv = calcInverse(sigma_lg)

// Second application of CPACK
sigma_cp2_inv= CPACK_M_I1_to_x1(left0,right0,
                sigma_cp,sigma_lg_inv)
sigma_cp2    = calcInverse(sigma_cp_inv)

// Reorder index arrays to implement
// mapping generated by lexGroup
left1        = remapArray_T_I0_to_I1(left0,
                sigma_lg)
right1       = remapArray_T_I0_to_I1(right0,
                sigma_lg)

// Adjust values in index arrays to
// reflect final data mapping
left2        = adjustIndexArray_R_x0_to_x2(
                left1,sigma_cp2)
right2       = adjustIndexArray_R_x0_to_x2(
                right1,sigma_cp2)

// Reorder data arrays to reflect
// final data mapping
x2           = remapArray_R_x0_to_x2(x0,
                sigma_cp2)
y2           = remapArray_R_x0_to_x2(y0,
                sigma_cp2)

// transformed simplified Moldyn code
do t = 1 to num_steps
   do i = 1 to num_nodes
      x2[i] = y2[i]
   enddo

   do j = 1 to num_inter
      y2[left2[j]] += x2[left2[j]]
                    - x2[right2[j]]
      y2[right2[j]] += x2[left2[j]]
                    - x2[right2[j]]
   enddo
enddo
```

Figure 5: Composed inspector for CPACK followed by Lexgroup

- Whereas run-time data and iteration reorderings are realized with inspectors which traverse the data mappings, sparse tiling inspectors traverse the data dependences between two loops (or one loop with self-dependences carried by an outer loop).
- Sparse tiling can also be used to provide parallelism which results in better performance than typical owner-computes methods [27].

In the simplified Moldyn example, applying sparse tiling after the CPACK, lexGroup, CPACK, series of run-time transformations described in section 3 can be specified with the following iteration-reordering transformation, where θ is the tiling function.

$$\begin{aligned} T_{I_1 \to I_2} &= \{[t,1,i,1] \to [t,\theta(1,i),1,i,1]\} \\ &\quad \cup [t,2,j,k] \to [t,\theta(2,j),2,j,k]\} \end{aligned}$$

The composition of lexGroup with sparse tiling is determined by applying $T_{I_1 \to I_2}$ to $T_{I_0 \to I_1}$.

Sparse tiling starts with a seed partitioning of iterations in one of the loops. If other data and iteration reordering transformations have been applied, a simple block partitioning of the iterations is sufficient. From this seed partitioning tiles are grown to the other loop in the pair by a traversal of the data dependences between the two loops.

Since the transformed code must traverse the final iteration space in lexicographical order, a schedule is created to indicate the iterations within each tile.

$$sched(b,q) = \{[i] \mid \theta(q,i) = b\}$$

The pseudo-code for the transformed version of simple Moldyn is shown in figure 6.

5. EXPERIMENTAL RESULTS

In our experiments we compare various compositions of run-time transformations on the MOLDYN, NBF, and IRREG benchmarks. The MOLDYN benchmark is taken from the molecular dynamics application CHARMM, the NBF kernel is taken from the GROMOS molecular dynamics code, and the IRREG kernel exhibits the types of computations found in partial differential equation solvers [9]. The sizes of the datasets in terms of nodes and edges in a representative graph are as follows.

Dataset	num nodes	num edges
mol1	131072	1179648
mol2	442368	3981312
foil	144649	1074393
auto	448695	3314611

The compositions consist of a data reordering transformation (CPACK or Gpart) followed by a iteration reordering transformation (bucket tiling [18], lexicographical grouping, or lexicographical sorting). With CPACK we try performing another round of CPACK data reordering followed by iteration reordering. Finally, we

```
// First application of CPACK
sigma_cp_inv = CPACK_M_I0_to_x0(left0,
                 right0)
sigma_cp     = calcInverse(sigma_cp_inv)

// Application of lexGroup
sigma_lg     = lexGroup_M_I0_to_x1(left0,
                 right0,sigma_cp)
sigma_lg_inv = calcInverse(sigma_lg)

// Second application of CPACK
sigma_cp2_inv= CPACK_M_I1_to_x1(left0,right0,
                 sigma_cp,sigma_lg_inv)
sigma_cp2    = calcInverse(sigma_cp_inv)

// Sparse Tiling
sched        = sparseTile_D_I1_to_I1(left0,
                 right0,sigma_lg)

// Reorder index arrays to implement
// mapping generated by lexGroup
left1        = remapArray_T_I0_to_I1(left0,
                 sigma_lg)
right1       = remapArray_T_I0_to_I1(right0,
                 sigma_lg)

// Adjust values in index arrays to
// reflect final data mapping
left2        = adjustIndexArray_R_x0_to_x2(
                 left1,sigma_cp2)
right2       = adjustIndexArray_R_x0_to_x2(
                 right1,sigma_cp2)

// Reorder data arrays to reflect
// final data mapping
x2           = remapArray_R_x0_to_x2(x0,
                 sigma_cp2)
y2           = remapArray_R_x0_to_x2(y0,
                 sigma_cp2)

// transformed simplified Moldyn code
do t = 1 to num_steps
   do b=1 to num_tiles
      do i in sched(b,1)
         x2[i] = y2[i]
      enddo

      do j in sched(b,2)
         y2[left2[j]] += x2[left2[j]]
                       - x2[right2[j]]
         y2[right2[j]] += x2[left2[j]]
                       - x2[right2[j]]
      enddo
   enddo
enddo
```

Figure 6: Composed inspector, including sparse tiling

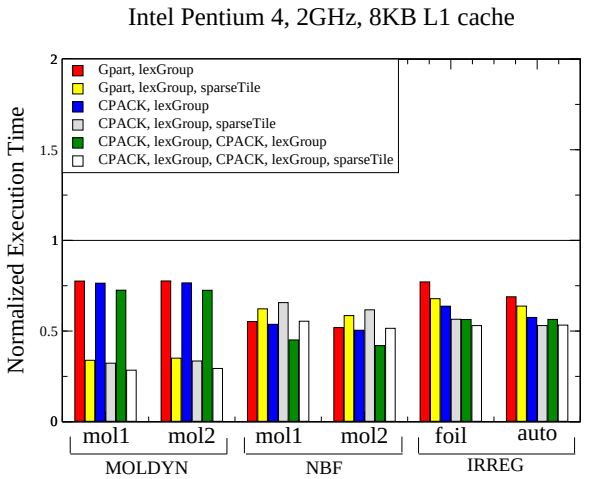

Figure 7: Normalized execution time without overhead, Pentium 4

perform full sparse tiling after the various compositions to see if it is possible to improve inter-loop locality.

The composed run-time inspectors were tested on two architectures: a 375MHz Power 3 (64KB L1 cache)[1] and a 2GHz Pentium 4 (8KB L1 cache). Of all the intra-loop iteration reordering transformations, lexicographical grouping consistently exibited the best performance to overhead tradeoff, therefore, due to space constraints the results in this paper use different data reordering transformations with lexicographical grouping and with or without sparse tiling.

Figures 7 and 8 show the normalized execution times for the various compositions without overhead. The number of iterations (time steps) in the outermost loop required to amortize the run-time overhead for both machines are shown in figures 9

When we apply our full sparse tiling technique in composition with existing run-time data and iteration reordering reordering techniques, we observe reduced execution time in the MOLDYN and IRREG benchmarks. Applying sparse tiling to the NBF benchmark after results in slow-downs. MOLDYN does more computation in all three loops within the time loop than the other 2 kernels. This results in more data reuse which takes better advantage of the temporal locality provided by sparse tiling. It is possible that NBF needs a different sparse tiling strategy. In these results only one iteration of the time loop was sparse tiled. Better results might occur when sparse tiling across 2 or 3 iteration of the time loop. IRREG experiences small improvements in execution time due to sparse tiling; however, sparse tiling affects the overhead in IRREG more than in MOLDYN.

[1]A single node of the IBM Blue Horizon at the San Diego Supercomputer Center.

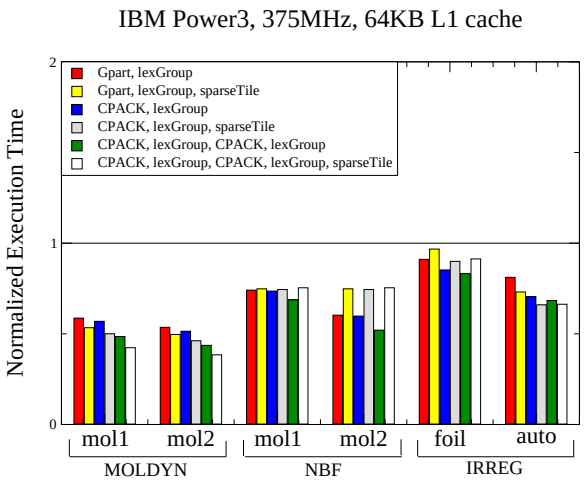

Figure 8: Normalized execution time without overhead, Blue Horizon

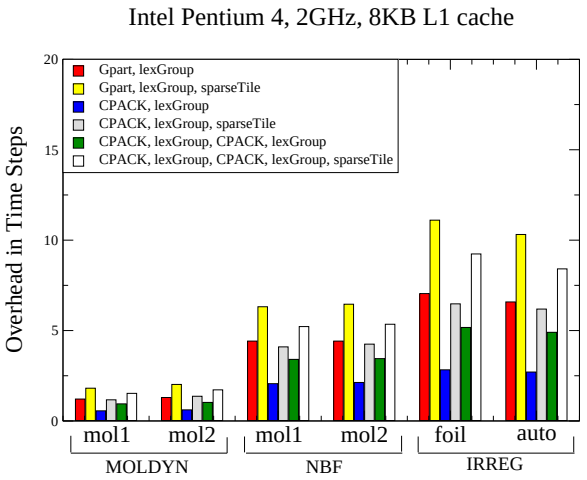

Figure 9: Overhead in time-steps, Pentium 4

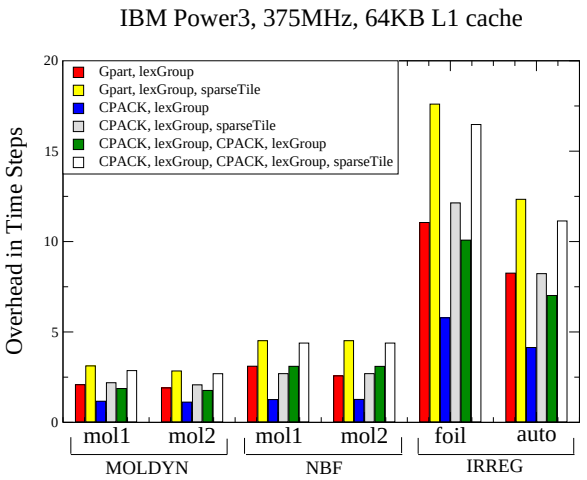

Figure 10: Overhead in time steps, Blue Horizon

6. RELATED WORK

Researchers have looked at extending data dependence analysis to run-time to handle non-affine memory references [20, 23]. These techniques typically attempt to disprove a dependence at runtime in order to exploit parallelism. Instead, our work requires the run-time traversal of dependences between loops.

Previous work has already explored how to generate code when the algorithm and the sparse data structure are specified separately [3] [22] [16] [10]. This work typically has the user specify the algorithm on a dense matrix. Since dense matrices are usually represented as two-dimensional arrays, this leads to algorithms that look very similar to algorithms presented in numerical analysis textbooks. We are describing the sparse data structures at a lower-level of abstraction, uninterpreted function symbols.

An obvious extension of our work is the automatic generation of specialized run-time inspectors and remappers. Specializing an inspector for a reordering transformation in the context of its position in a composition of reorderings should result in less overhead than an inspector implemented in a run-time library, since the latter must be generally applicable. The need for specialized inspectors has been described in run-time reordering transformations work for data locality [17] and parallelism [8].

Finally, to guide the selection of the both run-time and compile-time transformations in the same framework, it will be necessary to understand their interaction. This problem will be even more difficult than the static guidance problem. A heuristic for guiding a given data or iteration reordering might involve its own parameters (such as the seed partition size required by sparse tiling). The selection and order of run-time transformations will depend on information available at runtime as well as compile time. In the domain of data and iteration reordering, [19, 30] propose methods for guidance when some information such as the data acess pattern is not available until run-time.

7. CONCLUSIONS

Representing the effect of run-time data and iteration reordering transformations at compile-time is an important step towards the automatic generation of specialized inspectors and remappers. This paper shows how to use an existing compile-time framework to formally represent the changes in dependences and data mappings which occur when a number of data and iteration reorderings are performed. By showing that sparse tiling can be represented in our framework, we demonstrate its general applicability to other irregular codes; until now, it has been used only on Gauss-Seidel. Our framework suggests new compositions of these run-time transformations, and experimental results presented for the MOLDYN, NBF, and IRREG benchmarks show that significant improvements can be obtained, particularly for the molecular dynamics kernel. The greatest perfor-

mance improvements come from the ability to exploit both intra- and inter-loop locality.

8. ACKNOWLEDGEMENTS

This work was supported by an AT&T Labs Graduate Research Fellowship, a Lawrence Livermore National Labs LLNL grant, and in part by NSF Grant CCR-9808946. Equipment used in this research was supported in part by the National Partnership for Computational Infrastructure (NPACI). We used Rational PurifyPlus as part of the SEED program.

We would like to thank Hwansoo Han for making kernels and run-time inspector code available. We would also like to thank the students of CSE238 at UCSD for comments and suggestions.

9. REFERENCES

[1] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Conference Proceedings of the 2000 International Conference on Supercomputing*, pages 141–152, Santa Fe, New Mexico, May 2000.

[2] I. Al-Furaih and 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)*, pages 298–302, Los Alamitos, 30– 3 1998.

[3] Aart J. C. Bik and Harry A. G. Wijshoff. Compilation techniques for sparse matrix computations. In *Conference Proceedings, 1993 International Conference on Supercomputing*, pages 416–424, Tokyo, Japan, July 20–22, 1993. ACM SIGARCH.

[4] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality, November 1994.

[5] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. 24th National Conf. ACM*, pages 157–172, New York, 1969.

[6] Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 229–241, Atlanta, Georgia, May 1–4, 1999.

[7] Craig C. Douglas, Jonathan Hu, Markus Kowarschik, Ulrich Rüde, and Christian Weiß. Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transaction on Numerical Analysis*, pages 21–40, February 2000.

[8] E. Gutiérrez, R. Asenjo, O. Plata, and E. L. Zapata. Automatic parallelization of irregular applications. *Parallel Computing*, 26(13–14):1709–1738, 2000.

[9] Hwansoo Han and Chau-Wen Tseng. A comparison of locality transformations for irregular codes. In *5th International Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR'2000)*. Springer, 2000.

[10] J. Irwin, J. Loingtier, J. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming of sparse matrix code. In *Proceedings of the Scientific Computing in Object-Oriented Parallel Environments First International Conference (ISCOPE)*, pages 249–256. Springer, 1997.

[11] M. Kandemir, A. Choudhary, J. Ramanujam, and 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-98)*, pages 285–296, Los Alamitos, November 1998.

[12] Wayne Kelly and William Pugh. A unifying framework for iteration reordering transformations. Technical Report CS-TR-3430, University of Maryland, College Park, February 1995.

[13] Induprakas Kodukula and Keshav Pingali. Transformations for imperfectly nested loops. In ACM, editor, *Supercomputing '96 Conference Proceedings: November 17–22, Pittsburgh, PA*, pages ??–??, 1996.

[14] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal on Parallel Processing*, 22(2), April 1994.

[15] Lee-Chung Lu. A unified framework for systematic loop transformations, April 1991.

[16] Nikolay Mateev, Keshav Pingali, Paul Stodghill, and Vladimir Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *International Conference on Supercomputing*, Santa Fe, New Mexico, USA, May 2000.

[17] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 1999 Conference on Supercomputing*, pages 425–433, June 1999.

[18] Nicholas Mitchell, Larry Carter, and Jeanne Ferrante. Localizing non-affine array references. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 192–202, Newport Beach, California, October 12–16, 1999.

[19] Nick Mitchell, Larry Carter, and Jeanne Ferrante. A modal model of memory. In V.N.Alexandrov, J.J. Dongarra, and C.J.K.Tan, editors, *Proceedings of the 2001 International Conference on Computational Science*, Lecture Notes in Computer Science, May 28-30, 2001.

[20] Pugh and Wonnacott. Nonlinear array dependence analysis. Technical Report CS-TR-3372, Dept. of Computer Science, Univ. of Maryland, nov 1994.

[21] William Pugh and Evan Rosser. Iteration space slicing for locality. In *LCPC Workshop*, La Jolla, California, August 1999.

[22] William Pugh and Tatian Shpeisman. Sipr: A new framework for generating efficient code for sparse matrix computations, August 1998.

[23] Sivius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid analysis: Static and dynamic memory reference analysis. Technical report, Department of Computer Science, Texas A&M University, 2002.

[24] Vivek Sarkar and Radhika Thekkath. A general framework for iteration-reordering loop transformations. In Christopher W. Fraser, editor, *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 175–187, San Francisco, CA, June 1992.

[25] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John 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):118–141, June 1995.

[26] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Rescheduling for locality in sparse matrix computations. *Springer Lecture Notes in Computer Science*, 2073, 2001.

[27] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, Jonathan Freeman, and Barbara Kreaseck. Combining performance aspects of irregular gauss-seidel via sparse tiling. In *LCPC Workshop*, College Park, Maryland, July 2002.

[28] William Thies, Frederic Vivien, Jeffrey Sheldon, and Saman Amarasinghe. A unified framework for schedule and storage optimization. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, volume 36.5 of *ACM SIGPLAN Notices*, pages 232–242, June 2001.

[29] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 274–286, Paris, France, December 2–4, 1996.

[30] Hao Yu, Francis Dang, and Lawrence Rauchwerger. Parallel reductions: An application of adaptive algorithm selection. In *LCPC Workshop*, College Park, Maryland, July 2002.