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

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

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

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

ABSTRACT

Many important applications, such as those using sparse data structures, have memory reference patterns that are unknown at compile-time. Prior work has developed run-time reorderings of data and computation that enhance locality in such applications.

This paper presents a compile-time framework that allows the explicit composition of run-time data and iteration-reordering transformations. Our framework builds on the iteration-reordering framework of Kelly and Pugh to represent the effects of a given composition. To motivate our extension, we show that new compositions of run-time reordering transformations can result in better performance on three benchmarks.

We show how to express a number of run-time data and iteration-reordering transformations that focus on improving data locality. We also describe the space of possible run-time reordering transformations and how existing transformations fit within it. Since sparse tiling techniques 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. Finally, within the presented framework data need be remapped *only once* at runtime for a given composition thus exhibiting one example of overhead reductions the framework can express.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Optimization

General Terms

Performance, Experimentation

Keywords

optimization, run-time transformations, data remapping, iteration reordering, inspector/executor, sparse tiling

PLDI'03, June 9–11, 2003, San Diego, California, USA.

1. INTRODUCTION

Data locality and parallelism are essential for improving the performance of applications on current architectures. Data and loop transformations can further both goals. Until recently the focus has been primarily on compile-time transformation frameworks [19, 27, 18, 3, 16, 17, 32, 14, 31] restricted to affine loop bounds and affine array references. These frameworks allow for the uniform representation, the composition, the legality determination, and sometimes a benefit model of various compile-time transformations.

One such framework — that of Kelly and Pugh [16] — represents loop nests as iteration spaces. A compiler can use this framework to transform iteration spaces. The legality of a transformation is determined by the transformed data dependences of the program. Previous compile-time frameworks (including theirs) conservatively assume dependence when faced with non-affine memory references, which occur in many important applications such as sparse matrix and unstructured mesh computations [21]. Fortunately, the Kelly and Pugh framework describes non-affine memory references (such as indirect memory references `A[B[i]]`) by using Presburger arithmetic with uninterpreted function symbols [23]. We exploit this ability to specify *data mappings* between loop iterations and data locations, and *dependences* between loop iterations, when non-affine memory references are involved. We can also express run-time data and iteration-reordering transformations for locality, which include consecutive packing [7], graph partitioning [12], bucket-tiling [21], lexicographical grouping [7], full sparse tiling [29], and cache blocking [9].

Describing the effect of 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, the transformation legality checks provide constraints on the run-time reordering functions. Finally, the overhead involved in generating the run-time reordering functions can be reduced with various optimizations, such as moving the data to new locations only once and traversing fewer dependences.

Run-time reordering transformations are implemented with inspectors and executors, originally developed for parallelization [6]. In this setting, the *inspector* traverses the index arrays that describe the data mappings and/or dependences. Based on the values in these arrays, it generates data and/or iteration-reordering functions. The *executor* is a transformed version of the original loop that uses the reordered data and/or new schedule based on the reordering functions. Our

key insight is that given a composition of run-time reorderings, the modified data mappings and/or dependences are used by inspectors for subsequent run-time reordering transformations. While this paper focuses on data locality, our framework can also be used to describe run-time reordering transformations for parallelism.

Formalizing run-time data and iteration reorderings is only one step toward our goal of automating the creation of composite run-time transformations for computations with non-affine memory references. Still needed are methods of automatically generating run-time inspectors for a variety of reordering heuristics. Another key component is guidance mechanisms that decide when to apply which sequence of transformations. These decisions should probably be made at runtime based on the characteristics of the actual data mappings and dependences. Nevertheless, we believe that the framework presented here helps make such a system possible, and our experiments (on the `irreg`, `nbf`, and `moldyn` benchmarks [12]) illustrate that, in some cases, large performance improvements result from composite run-time reordering transformations.

Summarizing, we make the following contributions:

- 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 composing run-time reordering transformations.
- We show how to use an existing compile-time framework to describe a number of existing run-time data and iteration-reordering transformations that improve data locality. We also describe the space of possible run-time reordering transformations and how existing transformations fit within it.
- We show how sparse tiling techniques, which improve locality in loops with data dependences, 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 that show moving the data to new locations only once reduces the overhead of composed run-time reordering transformations.

Section 2 motivates the composition of run-time reordering transformations by describing experimental results where full sparse tiling composed with other transformations can result in significant performance improvements. Section 3 reviews the terminology for the Kelly and Pugh framework using an example irregular kernel (used throughout the paper). In Section 4, we describe how to formally express run-time reordering transformations and the space of possibilities for such transformations. Section 5 formally describes an example composed inspector. Section 6 shows examples of using the framework to reduce the overhead of run-time reordering transformations. Finally, sections 7, 8 and 9 are future work, related work, and conclusions.

2. MOTIVATION FOR COMPOSITIONS

Our experiments compare the performance resulting from various run-time transformation compositions on the `moldyn`, `nbf`, and `irreg` benchmarks. The goal is to motivate a framework that allows compile-time composition of run-time reordering transformations by showing that benchmarks with hand-coded composed transformations result in improved performance. The `moldyn` benchmark is taken from the molecular dynamics application CHARMM, the `nbf` benchmark is taken from the GROMOS molecular dynamics code, and the `irreg` benchmark exhibits the types of computations found in partial differential equation solvers [12].

```
do s = 1 to num_steps
   do i=1 to num_nodes
S1    x[i] = x[i] + vx[i] + fx[i]
   enddo

   do j=1 to num_inter
S2    fx[left[j]] += g(x[left[j]], x[right[j]])
S3    fx[right[j]] += g(x[left[j]], x[right[j]])
   enddo

   do k=1 to num_nodes
S4    vx[k] += fx[k]
   enddo
enddo
```

Figure 1: Simplified `moldyn` example

Figure 1 shows a simplified version of the `moldyn` benchmark. There is an outer time-stepping loop that makes amortization of run-time reordering overhead possible. Statement $S1$ calculates the new position of a molecule in the `x` coordinate using the old position, velocity, and acceleration. The `j` loop calculates the forces on the molecules using the `left` and `right` index arrays, which indicate interaction pairs. Previous work [7, 12] refers to `left` and `right` as the access or *index* arrays, and `x`, `vx`, and `fx` as base or *data* arrays.

Using the simplified `moldyn` benchmark in Figure 1, this section describes some existing run-time reordering transformations for improving the data locality within the `j` loop. We also describe the application of sparse tiling techniques, which improve locality between the `i`, `j`, and `k` loops. Finally, we present experimental results for all three benchmarks (`moldyn`, `nbf`, and `irreg`) on two different machines when various compositions of run-time reordering transformations are applied.

2.1 Run-time Data Reordering

Given a loop with non-affine memory references like the `j` loop in Figure 1, run-time data reordering transformations attempt to improve the spatial locality in the loop by reordering the data based on the order in which it is referenced in the loop. Consecutive packing (CPACK [7]) and graph partitioning (Gpart [12]) are two example data reordering transformations discussed in this paper.

Figure 2 shows one possible mapping of iterations in the `j` loop to locations in the data arrays `x` and `fx` based on values in the index arrays `left` and `right`. A CPACK data reordering would result in the new data mapping shown in Figure 3. Notice that the data has been reordered based on the memory reference order of the original mapping, therefore the computation exhibits better spatial locality.

We use the CPACK [7] and Gpart [12] data reordering transformations to reorder the `x`, `fx`, and `vx` arrays based

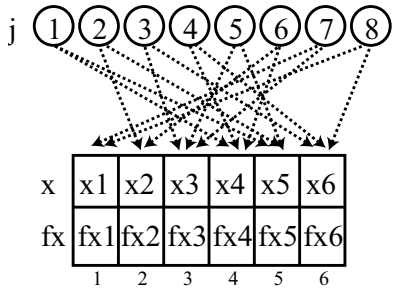

Figure 2: Example mapping of iterations in the `j` loop to locations in the data arrays `x` and `fx`. Here, circles represent iterations of the `j` loop inside one iteration of the outer time-stepping loop.

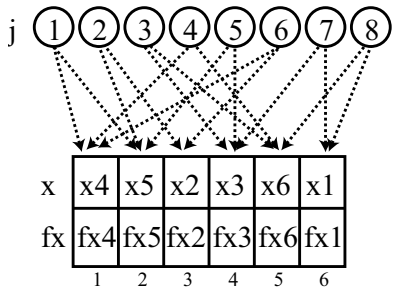

Figure 3: Example of Figure 2 mapping after the CPACK data reordering.

on the iteration to data mapping in the `j` loop.

Partitioning algorithms like Gpart [12] logically operate on a graph where each data location is a node. There is an edge between two nodes whenever their associated data is accessed within a loop iteration. By partitioning the nodes (ie. data) of the graph so that the data associated with each partition fits into (some level of) cache and ordering the data consecutively within a partition, Gpart improves the spatial locality of the computation.

2.2 Run-time Iteration Reordering

Often it is beneficial for an iteration reordering of a loop with non-affine memory accesses to follow a data reordering [7, 12]. The goal of iteration reordering is to reorder the iterations based on the order in which the loop touches the data.

One such iteration-reordering transformation is lexicographical grouping (lexGroup) [7]. For the simple `moldyn` example, Figure 3 shows how lexGroup further changes the data mapping between the iterations in the `j` loop to locations in the data arrays `x` and `fx`. Notice that iterations which touch the same or adjacent data locations now execute consecutively, therefore the computation exhibits better temporal and spatial locality.

We experimented with the iteration-reordering transformations bucket tiling [21] and lexicographical sorting [12] as well. However, lexicographical grouping (lexGroup) consistently exhibited the best performance to overhead trade-off on our benchmarks; therefore, the results in this paper always use lexGroup for reordering the iteration of the `j` loop.

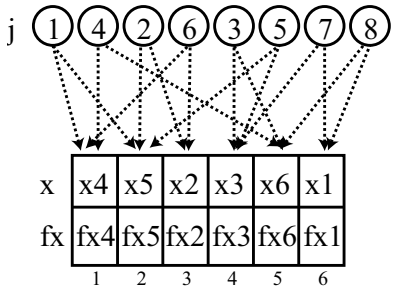

Figure 4: Example of Figure 2 mapping after the CPACK data reordering followed by a lexGroup iteration reordering.

2.3 Sparse Tiling

Sparse tiling programming techniques, full sparse tiling [29] and cache blocking [9], were developed for an important kernel used in Finite Element Methods, Gauss-Seidel. Sparse tiling results in run-time generated tiles or iteration slices [24] that cut between loops or across an outer loop and that only access a subset of the total data. By performing an iteration reordering based on sparse tiling, locality between loops or iterations of an outer loop improves.

Sparse tiling differs from other iteration-reordering transformations in four ways.

- Sparse tiling improves the locality between loops and across iterations of outer loops even when there are data dependences. Other run-time iteration-reordering transformations for data locality are not applicable when there are data dependences.
- Whereas existing run-time iteration reorderings for locality are realized with inspectors which traverse the data mappings, sparse tiling inspectors traverse the dependences.
- Until now, sparse tiling transformations 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 program can be determined.
- Sparse tiling can also be used to provide a coarser granularity of parallelism than other run-time reordering transformations for parallelism [30].

Sparse tiling starts with a seed partitioning of iterations in one of the loops (or in one iteration of an outer loop). If other data and iteration-reordering transformations have been applied to the loop being partitioned, then consecutive iterations in the loop have good locality and a simple block partitioning of the iterations is sufficient to obtain an effective seed partitioning. From this seed partitioning tiles are grown to the other loops involved in the sparse tiling by a traversal of the data dependences between loops (or between iterations of an outer loop). The main difference between full sparse tiling and cache blocking is how tile growth occurs. In cache blocking [9], the seed partitioning occurs on the first iteration of an outer loop and then tiles are grown by shrinking each partition for later iterations of that outer loop. The remaining iteration points are assigned to one tile. Full sparse tiling allows the seed partitioning to occur

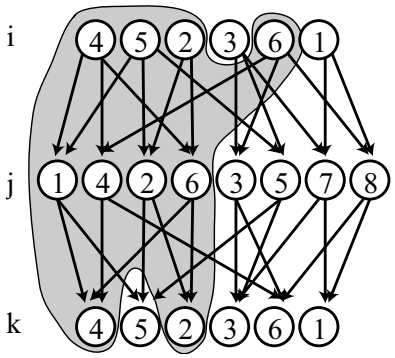

Figure 5: We highlight the iterations of one sparse tile for the code in Figure 1. The j loop has been blocked to provide a seed partitioning. In the full sparse tiled executor code, the iterations within a tile are executed atomically.

at any loop or iteration within an outer loop, and tiles are grown side-by-side.

For the simplified `moldyn` example, Figure 5 shows the status of the data dependences between iterations of the `i`, `j`, and `k` loops after applying the data reordering transformation CPACK and iteration-reordering transformation lexGroup. A full sparse tiling iteration reordering causes subsets of all three loops to be executed atomically as sparse tiles. Figure 5 highlights one such sparse tile where the `j` loop has been blocked to create a seed partitioning. Figure 14 shows the executor that iterates over tiles and then within the `i`, `j`, and `k` loops. Since iterations within all three loops touch the same or adjacent data locations, locality between the loops is improved in the new schedule.

In our experiments, we also apply a data reordering transformation, tile packing (tilePack), after applying full sparse tiling. TilePack reorders the data arrays based on how data is accessed within tiles. For example, in Figure 5 tilePack creates the data ordering $4, 2, 5, 6, 3, 1$, resulting in the consecutive ordering of data accessed within the highlighted tile.

2.4 Experimental Results

The sizes of the data sets we use in terms of nodes and edges in a representative graph are as follows.

Data set	nodes	edges
mol1	131072	1179648
mol2	442368	3981312
foil	144649	1074393
auto	448695	3314611

The baseline benchmarks and the executors for the run-time reordering transformation compositions use inter-array data regrouping [8] to leverage shared memory reference patterns between data arrays. All compositions we consider consist of a data reordering transformation (CPACK or Gpart) followed by the iteration-reordering transformation lexicographical grouping (lexGroup) for the `j` loop. We also perform the composition CPACK, lexGroup, CPACK, lexGroup. Finally, we apply full sparse tiling (FST) after the other compositions to see if improving the locality between the `i`, `j`, and `k` loops results in better performance.

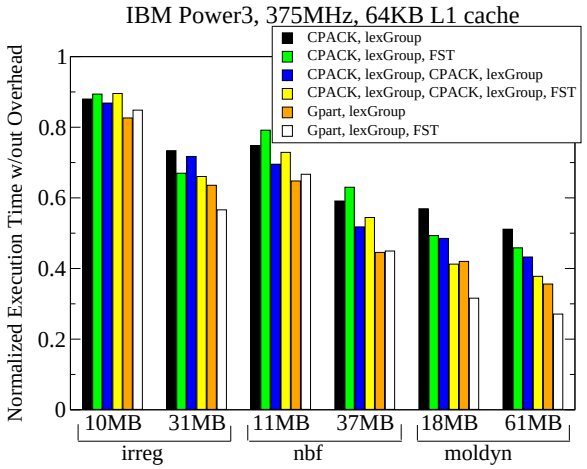

Figure 6: Normalized execution time without overhead on the Power3.

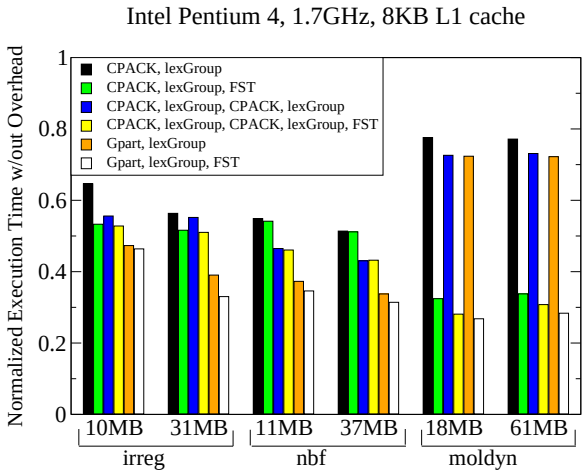

Figure 7: Normalized execution time without overhead on the Pentium 4.

The composed run-time reordering transformations are executed on two architectures: a 375MHz Power3 (64KB L1 cache)[1] and a 1.7GHz Pentium 4 (8KB L1 cache). On the Power3 we use the compilation command 'xlc -bmaxdata:0x80000000 -bmaxstack:0x10000000 -O3 -DNDEBUG', and on the Pentium 4 we use 'gcc -O2 -DNDEBUG'[2]. In our experiments, we target the L1 cache when selecting parameters for Gpart and full sparse tiling such as partition size.

Figures 6 and 7 show the normalized execution times for the various compositions without overhead. The number of outermost loop iterations (time steps) required to amortize the run-time overhead are shown in Figures 8 and 9. We calculate this number by taking the execution time of the inspector and dividing by the savings per time step observed in the executor.

When we apply our full sparse tiling technique in composition with existing run-time data and iteration-reordering

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

[2]gcc version 2.96 on Red Hat Linux 7.2 2.96-108.7.2

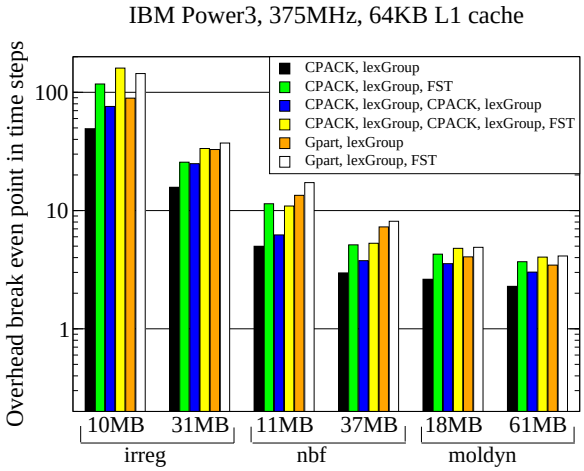

Figure 8: Amortization of the overhead on the Power3 in number of outer loop iterations based on the savings per iteration of the outer loop.

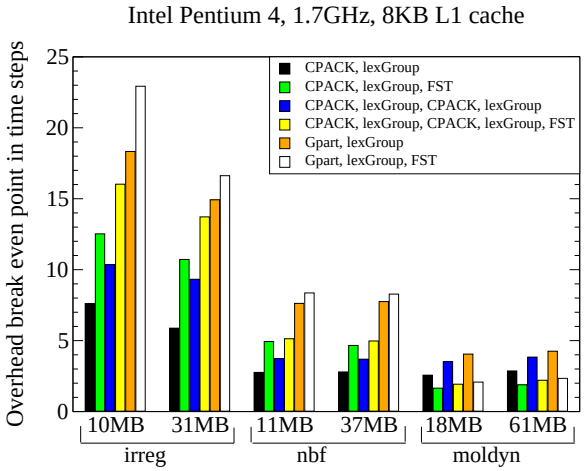

Figure 9: Amortization of the overhead on the Pentium 4 in number of outer loop iterations based on the savings per iteration of the outer loop.

transformations, we observe mixed results on the Power3. On the Pentium 4, using full sparse tiling in composition with the other data and iteration-reordering transformations results in improved performance for all our benchmarks and data sets. The results for the `moldyn` benchmark are especially impressive.

The `moldyn` benchmark does more computation and accesses more data than the other two benchmarks. In fact, for each molecule 72 bytes of data are stored. On the Pentium 4, the cache line is only 64 bytes long. Therefore, the data reordering transformations which improve spatial locality have less effect than iteration-reordering transformations like full sparse tiling. Full sparse tiling improves the performance in this benchmark to such an extent that it is actually easier to amortize the inspectors that include full sparse tiling (see Figure 9).

3. FRAMEWORK TERMINOLOGY

In this section, we review the Kelly-Pugh iteration-reordering framework terminology using the simplified `moldyn` code in Figure 1 as an example. Pugh and Wonnacott introduced the idea of using uninterpreted function symbols to statically describe the values in index arrays and other non-affine memory references [23]. Here we use this idea to describe some of the data mappings and dependences in the simplified `moldyn` example (Figure 1), which involve indirect memory references.

3.1 Loops, Statements, and Data

The traditional literature on loop transformations represents each iteration within a loop nest as an integer tuple, $\vec{p} = [p_1, ..., p_n]$, where p_q is the value of the iteration variable for the qth loop. Thus, a loop's *iteration space* is a set of integer tuples with constraints indicating the loop bounds.

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

This representation is not very convenient for representing transformations that operate on a collection of loops that are not perfectly nested. For instance, there are three traditional iteration spaces in the code shown in figure 1, and it is awkward to express how the sparse tiling run-time reordering transformation operates across all three. Ahmed et al. [1] and Kelly-Pugh [16] give two different methods for building a unified iteration space. In this paper, we use the Kelly-Pugh method. For the simplified `moldyn` example in Figure 1, they would use a four-dimensional space. Each loop 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. A program executes its iterations in lexicographic order of the unified iteration space.

For instance, using this representation, the $[s, k]$-th iteration of `S4` is denoted $[s, 3, k, 1]$ since `S4` is in the third statement (loop `j`) of the outer loop, and its the first statement within the `k` loop. The unified iteration space I_0 for the (untransformed) program is the following set:

$$\begin{aligned} I_0 = \quad & \{[s,1,i,1] \mid (1 \leq s \leq num\_steps) \\ & \wedge (1 \leq i \leq num\_nodes)\} \cup \\ & \{[s,2,j,q] \mid (1 \leq s \leq num\_steps) \\ & \wedge (1 \leq j \leq num\_inter) \\ & \wedge (1 \leq q \leq 2)\} \cup \\ & \{[s,3,k,1] \mid (1 \leq s \leq num\_steps) \\ & \wedge (1 \leq k \leq num\_nodes)\} \end{aligned}$$

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

3.2 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:

$$\begin{aligned} x_0 &= \{[m] \mid 1 \leq m \leq num\_nodes\} \\ vx_0 &= \{[m] \mid 1 \leq m \leq num\_nodes\} \\ fx_0 &= \{[m] \mid 1 \leq m \leq num\_nodes\} \\ left_0 &= \{[m] \mid 1 \leq m \leq num\_inter\} \\ right_0 &= \{[m] \mid 1 \leq m \leq num\_inter\} \end{aligned}$$

The subscripts "0" are used here since these are the data spaces for the original, untransformed program.

Define a *data mapping* $M_{I\to a}$ from iterations to sets of storage locations in an array a, so that for each iteration $\vec{p} \in I$, $M_{I\to a}(\vec{p})$ 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.

The `moldyn` example has the following data mappings:

$$\begin{aligned}
M_{I_0\to x_0} &= \{[s,1,i,1] \to [i]\} \\
&\quad \cup \{[s,2,j,q] \to [left(j)]\} \\
&\quad \cup \{[s,2,j,q] \to [right(j)]\} \\
&\quad \cup \{[s,3,k,1] \to [k]\} \\
M_{I_0\to fx_0} &= \{[s,1,i,1] \to [i]\} \\
&\quad \cup \{[s,2,j,1] \to [left(j)]\} \\
&\quad \cup \{[s,2,j,2] \to [right(j)]\} \\
&\quad \cup \{[s,3,k,1] \to [k]\} \\
M_{I_0\to vx_0} &= \{[s,1,i,1] \to [i]\} \\
&\quad \cup \{[s,3,k,1] \to [k]\} \\
M_{I_0\to left_0} &= \{[s,2,j,q] \to [j]\} \\
M_{I_0\to right_0} &= M_{I_0\to left_0}
\end{aligned}$$

Here we use uninterpreted function symbols to abstractly represent the data mappings for which Figure 2 shows concrete examples.

3.3 Dependences

Define the *dependences* $D_{I\to I}$ to be the set of directed edges between iterations $\vec{p} \in I$ that represent dependent computations. This set consists of the data dependence relations between statement `SV` and `SW` denoted as d_{VW}, with $1 \le V, W \le 4$ in the simplified `moldyn` example. For example, the dependences between statements `S1` ($[s,1,i,1]$), `S2` ($[s,2,j,1]$), and `S3` ($[s,2,j,2]$) due to the `x` and `fx` arrays are specified with the following dependence relation.

$$\begin{aligned}
d_{12} \cup d_{13} &= \{[s,1,i,1] \to [s',2,j,q] \mid (s \le s') \\
&\qquad \wedge (1 \le q \le 2) \\
&\qquad \wedge (i = left(j) \ \vee \ i = right(j))\}
\end{aligned}$$

The dependences between statements `S2` ($[s,2,j,1]$), `S3` ($[s,2,j,2]$), and `S4` ($[s,3,k,1]$) due to the `fx` arrays are specified with the following dependence relations.

$$\begin{aligned}
d_{24} \cup d_{34} &= \{[s,2,j,q] \to [s',3,k,1] \mid (s \le s') \\
&\qquad \wedge (1 \le q \le 2) \\
&\qquad \wedge (k = left(j) \ \vee \ k = right(j))\}
\end{aligned}$$

The arrows in Figure 5 represent concrete examples of these dependences. Notice that the dependences $d_{12} \cup d_{13}$ are symmetric to the dependences $d_{24} \cup d_{34}$ since both sets of dependences have constraints involving the `left` and `right` index arrays.

4. RUN-TIME REORDERING TRANSFORMATIONS

With run-time reordering transformations, an inspector traverses
mappings or data dependences transformed by reorderings produced by earlier inspectors. Thus, we want our framework to describe the data mappings and dependences in effect at all stages of the transformation process. At compile-time the data and iteration reorderings are expressed with uninterpreted function symbols. At run-time the inspectors traverse and generate index arrays to store the reordering functions.

Formally, a *data reordering transformation* is expressed with a 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 since data mappings do not affect data dependences - any one-to-one data remapping is legal. The result of remapping an array a is a new data mapping.

$$M_{I\to a'} = \{\vec{p} \to R_{a\to a'}(m) | m \in M_{I\to a}(\vec{p}) \wedge \vec{p} \in I\}$$

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

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 [15]:

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

The dependences of the transformed iteration space are

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

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

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

Given the new dependences and data mappings, we can plan further run-time transformations.

Within this framework run-time reordering transformations operate on subspaces within the unified iteration space. For each mapping of statements to unified iteration space a subspace can be specified by selecting a subset of dimensions in the mapping. A subspace is a candidate for run-time reordering transformations whenever the statements within the subspace involve non-affine memory references.

Data run-time reordering transformations are always legal since they do not affect dependences. It is not possible to perform iteration reordering if dependences between iterations in the subspace completely order the execution[3]. Some run-time iteration-reordering transformations (eg. lexicographical grouping, lexicographical ordering, bucket tiling)

[3]Reduction dependences are the exception, because they allow some reordering

can only be applied when there are no dependences between iterations in the selected subspace. When the subspace has dependences involving non-affine memory references, run-time iteration-reordering transformations such as run-time partial parallelization and sparse tiling satisfy the constraints ordained by the dependences by inspecting the dependences.

Run-time reordering transformations for partial parallelism traverse all the data dependences within an iteration subspace and create a run-time parallel schedule with maximal parallelism [25]. Parallelism is expressed within our framework by mapping parallel iterations to the same point in the unified iteration space.

Sparse tiling transformations partition a portion of the subspace and then grow tiles that respect the data dependences throughout the rest of the subspace. Since a loop is typically the portion of the subspace initially partitioned, sparse tiles are grown across dependences between loops or between iterations of an outer loop. By mapping all independent tiles to the same tile number, parallelism between tiles can be expressed.

5. COMPOSING TRANSFORMATIONS

This section illustrate how to specify the effects of applying several run-time data and iteration-reordering transformations for the simplified `moldyn` example.

5.1 Run-time Data Reordering

Run-time data reordering inspectors traverse data mappings and generate a data reordering function. Figure 10 shows the CPACK inspector code specialized for the original data mapping $M_{I_0 \to x_0}$ (specified in section 3.2) in the simplified `moldyn` example. This specialized CPACK inspector is called by the composed inspector in figure 11.

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`, `fx`, and `vx` arrays. Let $R_{x_0 \to x_1} = \{m \to m_1 \mid m_1 = \sigma_{cp}(m)\}$ specify the run-time data reordering on the `x` array, where x_0 is the data space for the `x` array in its original order, and x_1 is the data space for the reordered array `x1`. The new data mapping is specified as follows:

$$\begin{aligned} M_{I_0 \to x_1} \quad = \quad & \{[s,1,i,1] \to [\sigma_{cp}(i)]\} \\ & \cup \{[s,2,j,q] \to [\sigma_{cp}(left(j))]\} \\ & \cup \{[s,2,j,q] \to [\sigma_{cp}(right(j))]\} \\ & \cup \{[s,3,k,1] \to [\sigma_{cp}(k)]\} \end{aligned}$$

A data reordering σ_{gp} based on Gpart, orders data within the same partition consecutively. In the simple `moldyn` example, the abstract specification of the iteration to data mappings after applying Gpart is obtained by replacing σ_{cp} with σ_{gp}.

5.2 Run-time Iteration Reordering

If an iteration-reordering transformation on the `j` loop follows a data reordering transformation (as in our experiments), then the inspector for the iteration-reordering transformation will traverse the updated data mappings. In the simplified `moldyn` example, lexGroup will iterate over the data mappings that include the σ_{cp} function if lexGroup is performed after CPACK. The iteration reordering of the `i`, `j`, and `k` loops is specified as follows:

$$\begin{aligned} T_{I_0 \to I_1} \quad = \quad & \{[s,1,i,1] \to [s,1,i_1,1] \mid i_1 = \sigma_{cp}(i)\} \\ & \cup \{[s,2,j,q] \to [s,2,j_1,q] \mid j_1 = \delta_{lg}(j)\} \\ & \cup \{[s,3,k,q] \to [s,3,k_1,1] \mid k_1 = \sigma_{cp}(k)\} \end{aligned}$$

Since each iteration of the `i` and `k` loops directly maps to the `x`, `fx`, and `vx` arrays, the data reordering function generated for them, σ_{cp}, can be used for reordering the `i` and `k` loops as well. The transformation $T_{I_0 \to I_1}$ is legal because the only loop-carried dependences within the `i`, `j`, or `k` loops are reduction dependences between iterations of the `j` loop.

Due to the iteration reordering, the data mappings and dependences become:

$$\begin{aligned} M_{I_1 \to x_1} \quad = \quad & \{[s,1,\sigma_{cp}(i),1] \to [\sigma_{cp}(i)]\} \\ & \cup \{[s,2,\delta_{lg}(j),q] \to [\sigma_{cp}(left(j))]\} \\ & \cup \{[s,2,\delta_{lg}(j),q] \to [\sigma_{cp}(right(j))]\} \\ & \cup \{[s,3,\sigma_{cp}(k),q] \to [\sigma_{cp}(k)]\} \end{aligned}$$

$$(d'_{12} \cup d'_{13} \cup d'_{24} \cup d'_{34}) \subset D_{I_1 \to I_1}$$

$$\begin{aligned} & d'_{12} \cup d'_{13} = \{[s,1,\sigma_{cp}(i),1] \to [s',2,\delta_{lg}(j),q] \mid \\ & \quad (s \le s') \wedge (1 \le q \le 2) \wedge (i = left(j) \ \vee \ i = right(j))\} \\ & d'_{24} \cup d'_{34} = \{[s,2,\delta_{lg}(j),q] \to [s',3,\sigma_{cp}(k),1] \mid \\ & \quad (s \le s') \wedge (1 \le q \le 2) \wedge \ (k = left(j) \ \vee \ k = right(j))\} \end{aligned}$$

```
CPACK_M_I0_to_x0(left,right)
  // initialize alreadyOrdered bit vector
  // to all false
  count = 0
  do j=1 to num_inter
     mem_loc1 = left[j]
     mem_loc2 = right[j]

     if not alreadyOrdered(mem_loc1)
        sigma_cp_inv[count] = mem_loc1
        alreadyOrdered(mem_loc1) = true
        count = count + 1
     endif

     if not alreadyOrdered(mem_loc2)
        sigma_cp_inv[count] = mem_loc2
        alreadyOrdered(mem_loc2) = true
        count = count + 1
     endif
  enddo
  do i=1 to num_nodes
     if not alreadyOrdered(i)
        sigma_cp_inv[count] = i
        count = count + 1
     endif
  enddo
  return sigma_cp_inv
```

Figure 10: First CPACK inspector for `moldyn` called from composed inspector in Figure 11.

```
// First application of CPACK
sigma_cp_inv = CPACK_M_I0_to_x0(left,right)
sigma_cp     = calcInverse(sigma_cp_inv)

// First application of lexGroup
delta_lg     = lexGroup_M_I0_to_x1(left,right,
                   sigma_cp)
delta_lg_inv = calcInverse(delta_lg)

// Second application of CPACK
sigma_cp2_inv= CPACK_M_I1_to_x1(left,right,
                   sigma_cp,delta_lg_inv)
sigma_cp2    = calcInverse(sigma_cp2_inv)

// Second application of lexGroup
delta_lg2    = lexGroup_M_I1_to_x2(left,right,
                   sigma_cp,delta_lg_inv,sigma_cp2)
delta_lg2_inv= calcInverse(delta_lg2)

// Reorder data arrays to reflect
// final data mapping
x2           = remapArray_R_x0_to_x2(x,
                   sigma_cp, sigma_cp2)

// Adjust values in index arrays to
// reflect final data mapping
left         = adjustIndexArray_R_x0_to_x2(left,
                   sigma_cp,sigma_cp2)
right        = adjustIndexArray_R_x0_to_x2(right,
                   sigma_cp,sigma_cp2)

// Reorder index arrays to implement
// final iteration reordering
left2        = remapArray_T_I0_to_I2(left,
                   delta_lg,delta_lg2)
right2       = remapArray_T_I0_to_I2(right,
                   delta_lg,delta_lg2)
```

Figure 11: Composed inspector for CPACK, lexGroup, CPACK, and lexGroup composition.

5.3 Subsequent Transformations

When composing run-time reordering transformations, specialized instances of the relevant inspectors can be created that account for changes to the data mappings and dependences incurred by any previously planned inspectors. The explicit abstract description of how run-time reordering transformations affect each other allows new run-time reordering transformation compositions. For example, it is possible to generate another CPACK data reordering and lexGroup iteration reordering after generating the first CPACK and lexGroup reorderings.

Figure 12 shows how the second CPACK inspector is specialized to traverse the data mappings resulting from the first data and iteration-reordering functions, $M_{I_1 \to x_1}$. The array `delta_lg_inv` stores the inverse of the iteration reordering function δ_{lg}. The second CPACK inspector specifies a data reordering $R_{x_1 \to x_2} = \{m_1 \to m_2 | m_2 = \sigma_{cp2}(m_1)\}$.

A second iteration-reordering transformation for loop `j` traverses the data mapping $M_{I_1 \to x_2}$ and generates the reordering function δ_{lg2} to implement the transformation $T_{I_1 \to I_2}$.

```
CPACK_M_I1_to_x1(left,right,sigma_cp, sigma_lg_inv)
  // initialize alreadyOrdered bit vector
  // to all false
  count = 0
  do j1=1 to num_inter
     mem_loc1 = sigma_cp[left[delta_lg_inv[j1]]]
     mem_loc2 = sigma_cp[right[delta_lg_inv[j1]]]

     // same as CPACK_M_I0_to_x0 except creating
     // sigma_cp2_inv instead of sigma_cp_inv
  ...
  return sigma_cp2_inv
```

Figure 12: Second CPACK inspector for `moldyn` called from composed inspector in Figure 11.

```
do s = 1 to num_steps
   do i2 = 1 to num_nodes
      x2[i2] = x2[i2] + vx2[i2] + fx2[i2]
   enddo

   do j2 = 1 to num_inter
      fx2[left2[j2]] += g(x2[left2[j2]],
                          x2[right2[j2]])
      fx2[right2[j2]] += g(x2[left2[j2]],
                           x2[right2[j2]])
   enddo

   do k2 = 1 to num_nodes
      vx2[k2] += fx2[k2]
   enddo
enddo
```

Figure 13: Executor for simple `moldyn` example when inspector applies CPACK, lexGroup, CPACK, lexGroup composition.

$$
\begin{aligned}
M_{I_1 \to x_2} \;=\; & \{[s,1,\sigma_{cp}(i),1] \to [\sigma_{cp2}(\sigma_{cp}(i))]\} \\
& \cup \{[s,2,\delta_{lg}(j),q] \to [\sigma_{cp2}(\sigma_{cp}(left(j)))]\} \\
& \cup \{[s,2,\delta_{lg}(j),q] \to [\sigma_{cp2}(\sigma_{cp}(right(j)))]\} \\
& \cup \{[s,3,\sigma_{cp}(k),q] \to [\sigma_{cp2}(\sigma_{cp}(k))]\}
\end{aligned}
$$

$$
\begin{aligned}
T_{I_1 \to I_2} \;=\; & \{[s,1,i_1,1] \to [s,1,i_2,1] \mid i_2 = \sigma_{cp}(i_1)\} \\
& \cup \{[s,2,j_1,q] \to [s,2,j_2,q] \mid j_2 = \delta_{lg2}(j_1)\} \\
& \cup \{[s,3,k_1,q] \to [s,3,k_2,1] \mid k_2 = \sigma_{cp2}(k_1)\}
\end{aligned}
$$

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. This is done by manipulating reordering function arrays (`sigma_cp`, `delta_lg`, etc.) at run-time. The composed inspector in figure 11 remaps and updates the data and index arrays accordingly after all data and iteration reorderings have been computed.

```
do s = 1 to num_steps
   do t=1 to num_tiles
      do i4 in sched(t,1)
         x3[i4] = x3[i4] + vx3[i4] + fx3[i4]
      enddo

      do j4 in sched(t,2)
         fx3[left3[j4]] += g(x3[left3[j4]],
                             x3[right3[j4]])
         fx3[right3[j4]] += g(x3[left3[j4]],
                              x3[right3[j4]])
      enddo

      do k4 in sched(t,2)
         vx3[k4] += fx3[k4]
      enddo
   enddo
enddo
```

Figure 14: Sparse tiled executor when the composed inspector performs CPACK, lexGroup, CPACK, lexGroup, full sparse tiling, and tilePack.

5.4 Sparse Tiling

As iteration-reordering transformations, sparse tiling transformations can also be composed with other run-time reordering transformations. The main difference between sparse tiling transformations and other run-time reordering transformations for locality is that sparse tiling is applicable within subspaces of the unified iteration space that have data dependences. This is possible because sparse tiling inspectors traverse the data dependences.

In the simplified `moldyn` example, applying sparse tiling after the CPACK, lexGroup, CPACK, lexGroup series of run-time transformations described in Section 5.3 can be specified with the following transformation mapping.

$$\begin{aligned} T_{I_2 \to I_3} \quad = \quad & \{[s,1,i_2,1] \to [s,\theta(1,i_3),1,i_3,1] \mid i_3 = i_2\} \\ & \cup \{[s,2,j_2,q] \to [s,\theta(2,j_3),2,j_3,q] \mid j_3 = j_2\} \\ & \cup \{[s,3,k_2,1] \to [s,\theta(3,k_3),3,k_3,1] \mid k_3 = k_2\} \end{aligned}$$

The tiling function θ assigns a subspace of the unified iteration space to tile numbers. The subspace being sparse tiled in this example is $\{[1,i_2] \cup [2,j_2] \cup [3,k_2]\}$, with the seed partitioning occurring on the $[2,j_2]$ portion of the subspace. Figure 5 illustrates an instance of sparse tiled `moldyn` that uses full sparse tiling for tile growth.

In Figure 5, the computation exhibits better spatial locality if the data arrays are remapped after sparse tiling. Specifically if the data item (and corresponding iteration) numbered as 6 is put before 3, and 2 before 5, there is better locality. We refer to reordering the data and iterations in loops `i` and `k` based on the tiling function as *tile packing* (tilePack). TilePack uses an inspector that traverses the tiling function to generate a data reordering and iteration-reordering transformation.

```
// First application of CPACK
sigma_cp_inv = CPACK_M_I0_to_x0(left,right)
sigma_cp     = calcInverse(sigma_cp_inv)
// Reorder data arrays to reflect data mapping
x1           = remapArray_R_x0_to_x1(x,sigma_cp)
// Adjust values in index arrays
left         = adjustIndexArray_R_x0_to_x1(left,
                   sigma_cp)
right        = adjustIndexArray_R_x0_to_x1(right,
                   sigma_cp)

// First application of lexGroup
delta_lg     = lexGroup_M_I0_to_x1_B(left,right)
// Reorder index arrays to implement lexGroup
left1        = remapArray_T_I0_to_I1(left,
                   delta_lg)
right1       = remapArray_T_I0_to_I1(right,
                   delta_lg)

// Second application of CPACK
sigma_cp2_inv= CPACK_M_I1_to_x1_B(left1,right1)
sigma_cp2    = calcInverse(sigma_cp2_inv)
// Reorder data arrays to reflect data mapping
x2           = remapArray_R_x1_to_x2(x1,sigma_cp2)
// Adjust values in index arrays
left1        = adjustIndexArray_R_x1_to_x2(left1,
                   sigma_cp2)
right1       = adjustIndexArray_R_x1_to_x2(right1,
                   sigma_cp2)

// Second application of lexGroup
delta_lg2    = lexGroup_M_I1_to_x2_B(left1,right1)
// Reorder index arrays to implement
// final iteration reordering
left2        = remapArray_T_I1_to_I2(left1,
                   delta_lg2)
right2       = remapArray_T_I1_to_I2(right1,
                   delta_lg2)
```

Figure 15: Composed inspector for CPACK, lexGroup, CPACK, and lexGroup where data remapping and index array updates are done immediately after the relevant reordering function is generated.

$$\begin{aligned} R_{x_2 \to x_3} \quad = \quad & \{[m_2] \to [m_3] \mid m_3 = \sigma_{tp}(m_2)\} \\ T_{I_3 \to I_4} \quad = \quad & \{[s,t,1,i_3,1] \to [s,t,1,i_4,1] \mid i_4 = \sigma_{tp}(i_3)\} \\ & \cup \{[s,t,2,j_3,q] \to [s,t,2,j_4,q] \mid j_4 = j_3\} \\ & \cup \{[s,t,3,k_3,1] \to [s,t,3,k_4,1] \mid k_4 = \sigma_{tp}(k_3)\} \end{aligned}$$

Figure 14 shows the executor for the simple `moldyn` example when the iteration-reordering composition $T_{I_0 \to I_4}$ and the data reordering compositions $R_{x_0 \to x_3}$ are generated by composing the transformation mappings discussed in this section and previous sections. Since the transformed code must traverse the final iteration space in lexicographical order, a schedule (indexed by the tile and all but the last dimension in the subspace being sparse tiled) is created to

indicate the subset of iterations within each tile.

$$
\begin{aligned}
sched(t,1) &= \{[i_4] \mid i_4 = \sigma_{tp}(i_3) \wedge \theta(1,i_3) = t\} \\
sched(t,2) &= \{[j_4] \mid j_4 = j_3 \wedge \theta(2,j_3) = t\} \\
sched(t,3) &= \{[k_4] \mid k_4 = \sigma_{tp}(k_3) \wedge \theta(3,k_3) = t\}
\end{aligned}
$$

6. REDUCING THE OVERHEAD

The overhead of executing any inspector must be amortized to make run-time reordering transformations beneficial. In our experiments, we take advantage of the framework in two ways to generate efficient inspectors. First, in the benchmarks there are cases where two sets of data dependences satisfy the same constraints. Therefore, the full sparse tiling inspector need only traverse one set of data dependences while generating a legal tile function. Second, our experimental results indicate that remapping the data arrays after all run-time reordering functions have been generated reduces the execution time of inspectors that perform more than one data reordering.

Whenever two sets of data dependences satisfy the same constraints, it is only necessary to traverse one set at run-time. In the simplified `moldyn` example, the data dependences between statement `S1` and the statements in the `j` loop, `S2` and `S3`, are symmetric to the dependences between the statements in the `j` loop and statement `S4`. Therefore, our full sparse tiling inspector need only traverse one set of these dependences to grow the tiles from a seed partitioning of the `j` loop to the `i` and `k` loops. A similar situation occurs in all the benchmarks we use for experiments.

The framework allows the compiler to choose when to remap a data array. Figure 11 illustrates a composed inspector that performs data remapping and index array adjustments after all reordering functions are generated, and Figure 15 shows an inspector performing the same composition of transformations, but remapping and adjusting after each reordering function is generated. Notice that many of the functions, like `CPACK_M_I1_to_x1_B`, take fewer parameters in Figure 15 than in Figure 11. Since the index arrays `left` and `right` are remapped and adjusted after every transformation in Figure 15, the index arrays which maintain the reordering functions are not needed. This results in fewer indirect memory references in the composed inspector and can have an affect on its performance.

Our experience suggests that remapping and adjusting the index arrays after each transformation and remapping the data arrays after all data reordering transformations leads to the most efficient inspectors. Figure 16 shows the percentage execution time reduction when the data arrays are remapped after all transformations. The results are shown only for the `irreg` and `moldyn` benchmarks because `nbf` does not benefit from the tilePack data reordering transformation. Therefore, most of the compositions involving `nbf` do not use two or more data reordering transformations.

Optimal generation of composed inspectors is an open question. Our framework allows the expression of diverse possibilities.

7. FUTURE WORK

An obvious extension of our work is the automatic generation of specialized run-time inspectors. Specializing an inspector for a reordering transformation in the context of its position in a composed inspector 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 work for data locality [20] and parallelism [11].

Automatically generating code for the inspector and executor can leverage the work in [7], which describes compiler support for dynamic data packing, and the work in [16], which generates optimized code for compile-time transformations. Specifically, the techniques described in [16] can be used to generate the transformed executor code and the inspector code that traverses the data mappings and dependences. Automatically generating specialized and optimized versions of the various data and iteration-reordering algorithms such as CPACK, lexGroup, and full sparse tiling will be more challenging.

To completely automate the usage of composed run-time reordering transformations, more performance modeling work is needed to select between various compositions and the

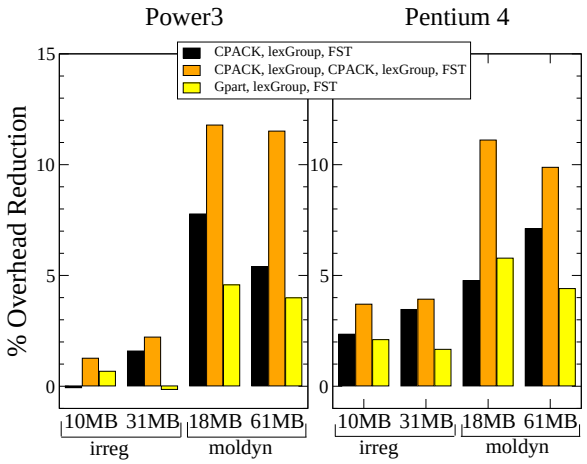

Figure 16: Percent reduction in inspector overhead for compositions with two or more data reorderings and data is only remapped once.

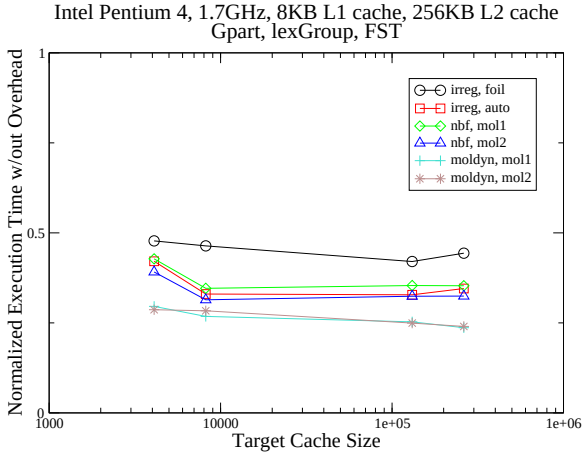

Figure 17: Normalized execution time without overhead on the Pentium 4 as the Gpart and full sparse tiling parameters are selected to target half of L1 cache, L1 cache, 1/2 of L2 cache, and L2 cache.

parameters for compositions. Figure 17 shows how the performance of the executor differs as the Gpart and full sparse tiling parameters are selected to target different cache sizes. The performance varies depending on the benchmark and dataset. The machine is also a factor. Since characteristics of the dataset are not available until runtime, the selection and order of run-time reordering transformations depend on information available at runtime as well as compile time. In the domain of data and iteration reordering, [22, 33] propose methods for guidance when some information such as the data access pattern is not available until runtime.

8. RELATED WORK

Run-time reordering transformations differ from dynamic compilation techniques such as those described in [10], because reordering transformations do not change the code at runtime. Instead the code has already been transformed and inspectors create data and iteration-reordering functions, which are stored in index arrays.

Researchers have developed run-time data dependence analysis to handle non-affine memory references [23, 26]. In [23] constraints for disproving dependences are evaluated at run-time. Rus et al. [26] take this further adding the ability to traverse all data dependences at run-time if necessary. They perform a hybrid (static and dynamic) data dependence analysis inter-procedurally. As we have described in this paper, traversing data dependences at run-time is necessary for some run-time reordering transformations.

Many run-time data reordering transformations [4, 2, 21, 7, 12] fit within our framework. Space filling curves and register tiling for sparse matrix vector multiply are two types of data reordering transformations that are more specialized. Data reorderings generated from space-filling curves [28, 20] traverse data mappings and mappings of data to spatial coordinates. The programmer must specify how data maps to spatial coordinates, therefore, such data reorderings can not be fully automated. Im and Yelick [13] have developed the SPARSITY code generator that improves the locality for the $\vec{x}$ and $\vec{b}$ vectors in the sparse matrix-vector multiplication $A\vec{x} = \vec{b}$. The dynamic register blocking techniques are useful for many application domains that use sparse matrices, but the work focuses on a single algorithm.

There has been a definite progression toward complete automation of run-time reordering transformations. Initially such transformations were incorporated into applications manually for parallelism [5]. Next, libraries with run-time transformation primitives were developed so that a programmer or compiler could insert calls to such primitives [6]. Currently, there are many run-time reordering transformations for which a compiler can automatically analyze and generate the inspectors [25, 7, 21, 12]. However, each transformation or composition of transformations are treated separately. Our framework provides a uniform representation for these transformations and describes how to compose any number of them at compile-time.

9. CONCLUSIONS

This paper motivates compositions of run-time data and iteration reordering transformations with experimental results showing significant performance improvements for the `moldyn`, `nbf`, and `irreg` benchmarks. We show how to use an existing compile-time framework to formally express the changes in dependences and data mappings which occur when a composition of data and iteration reorderings are performed. Representing the abstract effect of run-time data and iteration-reordering transformations at compile-time is an important step toward the automatic generation of specialized inspectors. 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. We also use two different optimizations to improve the performance of our composed inspectors, thus reducing the overhead of composed run-time reordering transformations.

10. ACKNOWLEDGMENTS

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 the benchmarks and run-time inspector code available. We would also like to thank the students of CSE238 at UCSD and anonymous reviewers for comments and suggestions.

11. REFERENCES

[1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 141–152, 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*, pages 298–302, Los Alamitos, March 30–April 3 1998.

[3] S. Carr, K. S. McKinley, and C. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, November 1994.

[4] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th National Conference ACM*, pages 157–172, 1969.

[5] R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives, AIAA-92-0562. In *Proceedings of the 30th Aerospace Sciences Meeting*, 1992.

[6] R. Das, M. Uysal, J. Saltz, and Yuan-Shin S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–478, 1994.

[7] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices, pages 229–241, May 1–4, 1999.

[8] C. Ding and K. Kennedy. Inter-array data regrouping. In *Twelfth International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, August 1999.

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

[10] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000.

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

[12] H. Han and C. Tseng. A comparison of locality transformations for irregular codes. In *5th International Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*. Springer, 2000.

[13] E. Im and K. Yelick. Optimizing sparse matrix computations for register reuse in sparsity. In V.N.Alexandrov, J.J. Dongarra, B.A.Juliano, R.S.Renner, and C.J.K.Tan, editors, *Computational Science - ICCS 2001*, Lecture Notes in Computer Science, pages 127–136. Springer, May 28-30, 2001.

[14] 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*, pages 285–296, November 1998.

[15] W. Kelly and W. Pugh. Finding legal reordering transformations using mappings. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, volume 892 of *Lecture Notes in Computer Science*, pages 107–124. Springer, August 8–10, 1994.

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

[17] Induprakas Kodukula and Keshav Pingali. Transformations for imperfectly nested loops. In *Proceedings of 1996 Conference on Supercomputing*. ACM Press, 1996.

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

[19] Lee-Chung Lu. A unified framework for systematic loop transformations. In *Proceedings of the 3rd Annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 28–38, 1991.

[20] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 1999 International Conference on Supercomputing*, ACM SIGARCH, pages 425–433, June 20–25 1999.

[21] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 192–202, October 12–16, 1999.

[22] N. Mitchell, L. Carter, and J. Ferrante. A modal model of memory. In V.N.Alexandrov, J.J. Dongarra, B.A.Juliano, R.S.Renner, and C.J.K.Tan, editors, *Computational Science - ICCS 2001*, Lecture Notes in Computer Science. Springer, May 28-30, 2001.

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

[24] W. Pugh and E. Rosser. Iteration space slicing for locality. In *Proceedings of the 12th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, August 1999.

[25] L. Rauchwerger, N. M. Amato, and D. A. Padua. Run-time methods for parallelizing partially parallel loops. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, 1995.

[26] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: Static & dynamic memory reference analysis. In *Proceedings of the 16th Annual ACM International Conference on Supercomputing*, June 2002.

[27] V. Sarkar and R. Thekkath. A general framework for iteration-reordering loop transformations. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices, pages 175–187, June 1992.

[28] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and 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):118–141, June 1995.

[29] M. M. Strout, L. Carter, and J. Ferrante. Rescheduling for locality in sparse matrix computations. In V.N.Alexandrov, J.J. Dongarra, B.A.Juliano, R.S.Renner, and C.J.K.Tan, editors, *Computational Science - ICCS 2001*, Lecture Notes in Computer Science. Springer, May 28-30, 2001.

[30] M. M. Strout, L. Carter, J. Ferrante, J. Freeman, and B. Kreaseck. Combining performance aspects of irregular gauss-seidel via sparse tiling. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, July 2002.

[31] W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe. A unified framework for schedule and storage optimization. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices, pages 232–242, June 2001.

[32] M. E. Wolf, D. E. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 274–286, December 2–4, 1996.

[33] H. Yu, F. Dang, and L. Rauchwerger. Parallel reductions: An application of adaptive algorithm selection. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, July 2002.