Compiler Supported High-level Abstractions for Sparse Disk-Resident Datasets [*]

Renato Ferreira* Gagan Agrawal† Joel Saltz*
*Department of Computer Science
University of Maryland, College Park MD 20742
{renato,saltz}@cs.umd.edu
†Department of Computer and Information Sciences
University of Delaware, Newark DE 19716
agrawal@cis.udel.edu

Abstract

Processing and analyzing large volumes of data plays an increasingly important role in many domains of scientific research. The complexity and irregularity of datasets in many domains make the task of developing such processing applications tedious and error-prone.

We propose use of high-level abstractions for hiding the irregularities in these datasets and enabling rapid development of correct, but not necessarily efficient, data processing applications. We present two execution strategies and a set of compiler analysis techniques for obtaining high performance from applications written using our proposed high-level abstractions. Our execution strategies achieve high locality in disk accesses. Once a disk block is read from the disk, all iterations that read any of the elements from this disk block are performed. To support our execution strategies and improve the performance, we have developed static analysis techniques for: 1) computing the set of iterations that access a particular right-hand-side element, 2) generating a function that can be applied to the meta-data associated with each disk block, for determining if that disk block needs to be read, and 3) performing code hoisting of conditionals.

We present experimental results from a prototype compiler implementing our techniques to demonstrate the effectiveness of our approach.

1 Introduction

Analysis and processing of large disk-resident multi dimensional scientific datasets is an important component of science and engineering. An increasing number of applications make use of very large multidimensional datasets. Examples of such datasets include raw and processed sensor data from satellites [31], output from hydrodynamics and chemical transport simulations [25], and archives of medical images [2].

Many of these domains involve complex and sparse datasets. For example, the dataset captured by a satellite can be viewed as a sparse three dimensional array, where time, latitude, and longitude are the three dimensions. Pixels for several, but not all, time values are available for any given latitude and longitude. Similarly, in a typical dataset available with a multi-grid virtual microscope, the same portion of a slide may be available at different resolution levels, but the entire slide is not captured at all resolution levels.

Developing applications that can efficiently process such disk-resident datasets is a challenging task. In domains like satellite data processing and virtual microscope, the complexity and irregularity of the dataset adds to the difficulty associated with developing and performance tuning any application that frequently accesses disk-resident data.

High-level abstractions of datasets can significantly reduce the complexity of application development. If the low-level details of the dataset are hidden from a programmer, a correct application can be developed with ease. However, the efficiency, in general, and the locality, in particular, of the code written in this fashion is likely to be poor. This places an additional burden on the compiler. The compiler, which has access to the low-level description of the data layout, can perform aggressive transformations, and achieve locality and high performance from the codes written by programmers not having access to the low-level data layout.

In this paper, we present two execution strategies and a set of compiler analysis techniques to support high-level abstractions of disk-resident datasets. We use a data parallel dialect of Java for expressing computations over sparse disk-resident datasets. In the first (basic) execution strategy we support, the right-hand-side data is read one disk block at a time. Once this data is brought into the memory, corresponding iterations are performed. The resulting code has a very high locality in disk accesses, but may also have extra computation and evaluation of conditionals in every iteration. We present two analysis techniques for supporting this execution strategy:

- A technique for computing the set of iterations that access a particular element of a collection.
- A static technique for generating a function that can be applied to the meta-data associated with each disk block, for determining if that disk block needs to be read.

We also present two techniques for improving the performance achieved from our basic execution strategy:

---

[*] This work was supported by NSF grant ACR-9982087, NSF CAREER award ACI-9733520, and NSF grant CCR-9808522.

- A variation of our basic execution strategy that exploits the regularity in the data elements comprising a disk block, to reduce some of the overheads associated with the execution strategy.
- An analysis framework for code motion of conditionals which further improves the performance data intensive codes using our execution strategy.

Our techniques have been implemented in a prototype compiler using the Titanium infrastructure from Berkeley. We present experimental results from three codes: a satellite data processing code, a multi-grid virtual microscope application, and a regular virtual microscope. Our experimental results on disk-resident datasets demonstrate the following:

- Our techniques are effective in producing efficient versions of these codes. Even with the high volume of communication, the speedups are quite impressive.
- The choice of execution strategy is crucial for obtaining high performance.
- The conditional code motion optimization achieves significant performance improvements.

The rest of the paper is organized as follows. The high-level data abstractions that we support are presented with a satellite data processing example in Section 2. Basic compilation and loop execution strategies are presented in Section 3. Three advanced compiler analysis techniques that support code generation and execution are presented in Section 4. Experimental results are presented in Section 5. We compare our work with related research efforts in Section 6 and conclude in Section 7. An additional example application, the multi-grid virtual microscope, is presented in the appendix.

2 Example Application and High-level Abstractions

In this section, we introduce our high-level abstractions for facilitating rapid development of applications that process disk-resident datasets. We use the satellite data processing application [11] as an example. We first describe the nature of the datasets captured by the satellites orbiting the earth, then describe the typical processing on these, and finally explain the high-level abstractions and data parallel language constructs we support for performing such processing.

2.1 Application Description: Satellite Data Processing

A satellite orbiting the earth collects data as a sequence of *blocks*. The satellites contain sensors for five different bands. The measurements produced by the satellite are short values (16 bits) for each band. As the satellite orbits the earth, the sensors sweep the surface building scan lines of 408 measurements each. Each block consists of 204 half scan lines, i.e., it is a 204×204 array with 5 short integers per element. Latitude, longitude, and time is also stored within the block for each measure.

The typical computation on this satellite data is as follows. A portion of earth is specified through latitudes and longitudes of end points. A time range (typically 10 days to one year) is also specified. For any point on the earth within the specified area, all available pixels within that time period are scanned and the *best* value is determined. Typical criteria for finding the *best* value is cloudiness on that day, with the least cloudy image being the best. The best pixel over each point within the area is used to produce a composite image. This composite image is used by researchers to study a number of properties, like deforestation over time, pollution over different areas, etc [31].

There are two sources of sparsity and irregularity in the dataset and computation. First, the pixels captured by the satellite can be viewed as a sparse three dimensional array, where time, latitude, and longitude are the three dimensions. Pixels for several, but not all, time values are available for any given latitude and longitude. The second source of irregularity in the dataset comes because the earth is spherical, whereas the satellite sees the area of earth it is above as a rectangular grid. Thus, the translation from the rectangular area that the satellite has captured in a given band to latitudes and longitudes is not straight forward.

2.2 Low Level Data Layout

In Figure 1, we show the essential structure associated with the satellite data processing application [11, 12].

The class `block` represents the data captured in each time unit by the satellite. This class has one function (`getData`) which takes a (latitude, longitude) pair and sees if there is any pixel in the given block for that location. If so, it returns that pixel. The class `SatOrigData` stores the data as a one dimensional array of blocks.

Classes `block` and `SatOrigData` are not visible to the programmer writing the processing code. The goal is to provide a simplified view of the dataset to the application programmers, thereby easing the development of correct, but not necessarily efficient, data processing application. The compiler translating the code obviously has the access to the source code of these classes, which enables it to generate efficient low-level code.

2.3 High Level Abstractions

The class `SatData` is the interface to the input dataset visible to the programmer writing the main execution code. Through its access function `getData`, this class gives the view that a 3-dimensional grid of pixels is available. The function `getData` has a special significance for the compiler, as we will discuss later.

The main processing function takes 6 command line arguments as the input. The first two specify a time range over which the processing is performed. The next four are the latitudes and longitudes for the two end points of the rectangular output desired. We need to iterate over all the blocks within the time range, examine all pixels which fall into the output region, and then perform the reduction operation (i.e. choosing the best pixel).

We specify this computation in a data parallel language as follows. We consider an abstract 3-dimensional rectangular grid, with time, latitude, and longitude as the three axes. This grid is abstract, because pixels actually exist for only a small fraction of all the points in this grid. However, the high-level code just iterates over this grid in the `foreach` loop. For each point q in the grid, which is a `(time, lat, long)` tuple, we examine if the block `SatData[time]` has any pixel. If such a pixel exists, it is used for performing a reduction operation on the object `OutImage[(lat,long)]`.

We have used a number of data parallel constructs in our language. These have also been used in object-oriented parallel systems like Titanium [38], HPC++ [7], and Concurrent Aggregates [13, 33], and are not unique to our approach. These constructs are:

```
Interface Reducinterface {
   {* Any object of a class implementing *}
   {* this interface is a reduction variable *}
}
public class pixel {
   short bands[5] ;
   short geo[2] ;
}
class block {
   short time ;
   pixel bands[204*204] ;
   pixel getData(Point[2] p) {
      {* Search for the (lat, long) on geo data *}
      {* Return the pixel if it exists *}
      {* else return null *}
   }
}
{* Low-level Data Layout *}
class SatOrigData {
   block[1d] data ;
   void SatOrigData(RectDomain[1] InputDomain) {
      data = new block[InputDomain] ;
   }
   pixel getData(Point[3] q) {
      Point[1] time = (q.get(0));
      Point[2] p = (q.get(1), q.get(2));
      return data[time].getData(p) ;
   }
}
public class OutputData
                 implements Reducinterface {
   void Accumulate(pixel input ) {
      {* Aggregate value of input *}
      {* pixel into output *}
   }
}
```

```
{* High-level Data Layout *}
public class SatData {
   SatOrigData data;
   void SatData(RectDomain[1] InputDomain) {
      data = new SatOrigData(InputDomain) ;
   }
   pixel getData(Point[3] q) {
      return data.getData(q) ;
   }
}
public class SatelliteApp {
   Point[1] minpt = ...
   Point[1] maxpt = ...
   RectDomain[1] InputDomain = [minpt : maxpt];
   SatData InputData = new Satdata(InputDomain) ;
   public static void main(int[] args) {
      Point[2] lowend = (args[2],args[4]);
      Point[2] highend = (args[3],args[5]);
      Rectdomain[2] OutputDomain = [lowend : highend];
      Point[3] low = (args[0], args[2], args[4]);
      Point[3] high = (args[1], args[3], args[5]);
      Rectdomain[3] AbsDomain = [low : high];
      Image[2d] OutImage = new OutputData[OutputDomain];

      foreach (Point[3] q in AbsDomain) {
         Point[2] p = (q.get(1), q.get(2));
         if (pixel val = satdata.getData(q))
            OutImage[p].Accumulate(val.data);
      }
   }
}
```

Figure 1: A Satellite Data-Processing Code

- A *rectdomain* is a collection of objects of the same type such that each object in the collection has a *coordinate* associated with it, and this coordinate belongs to a pre-specified rectilinear section.
- The *foreach* loop, which iterates over objects in a rectdomain, and has the property that the order of iterations does not influence the result of the associated computations.
- We use a Java interface called *Reducinterface*. Any object of any class implementing this interface acts as a *reduction variable* [18]. A reduction variable has the property that it can only be updated inside a *foreach* loop by a series of operations that are associative and commutative. Furthermore, the intermediate value of the reduction variable may not be used within the loop, except for self updates.

2.4 Discussion

The code, as specified above, can lead to very inefficient execution for at least two reasons. First, if the lookup is performed for every point in the abstract grid, it will have a very high overhead. Second, if the order of iterations in the loop is not carefully chosen, the locality can be very poor.

However, a very simplified view of the dataset has been provided to the programmer, facilitating rapid development of correct, but not necessarily efficient, data processing code. The programmer can view the available data as being associated with an abstract grid and an access to an abstract 3-dimensional array is translated into an access of 1-dimensional array of pixels.

3 Execution Strategies and Compilation Overview

This section gives an overview of the compilation and execution strategy we use. Details of advanced compilation techniques used in generating code and supporting such execution are presented in the next section.

The main challenge in executing a data intensive loop comes from the fact that the amount of data accessed in the loop exceeds the main memory. While the virtual memory support can be used for correct execution, it leads to very poor performance. Therefore, it is compiler's responsibility to perform memory management, i.e., determine which portions of output and input collections are in the main memory during a particular stage of the computation.

Based upon our experience from data intensive applications and developing runtime support for them [11, 10], the basic code execution scheme we use is as follows. The output data structure is divided into *tiles*, such that each tile fits into the main memory. The input dataset is read one disk block at a time. This is because the disks provide the highest bandwidth and incur the lowest overhead while ac-

```
foreach(r $\in \mathcal{R}$) {
    $O_1[\mathcal{S}_L(r)] = \mathcal{F}_1(O_1[\mathcal{S}_L(r)], I_1[\mathcal{S}_{R1}(r)], \ldots, I_n[\mathcal{S}_{Rn}(r)])$
    ...
    $O_m[\mathcal{S}_L(r)] = \mathcal{F}_m(O_1[\mathcal{S}_L(r)], I_1[\mathcal{S}_{R1}(r)], \ldots, I_n[\mathcal{S}_{Rn}(r)])$
}
```

Figure 2: Canonical Form of Loop: Output Collections Do Not Fit in Main Memory

cessing all data from a single disk block. Once an input disk block is brought into main memory, all iterations of the loop which read from this disk block and update an element from the current tile are performed. A tile from the output data structure is never allocated more than once, but a particular disk block may be read to contribute to multiple output tiles.

The challenges for the compiler, and the associated runtime support, in generating code and supporting such an execution strategy on a uniprocessor are as follows:

C1: Partitioning the output data structure into tiles fitting in main memory, as described above.

C2: For each tile, finding the disk blocks corresponding to each input collection that contribute to the value of output elements in that tile.

C3: Choosing an order for reading these disk blocks, to minimize disk seek time and achieve high throughput from the disks.

C4: For each such disk block identified, determining the elements in the disk block that are accessed in the loop, and the iteration(s) of the loop in which they are accessed.

C5: Performing the computation associated with each iteration of the loop, using elements from different input collections.

In addition, if the code is executed on a distributed memory parallel machine, the following challenges also arise:

P1: Work distribution, i.e., partitioning the iterations in the loop between processors.

P2: Handling communication between the processors.

We assume that disk-resident datasets have already been partitioned among the nodes, so data distribution decisions are not made by the compiler. Compiler and runtime analysis performed for ensuring correct and efficient communication for our target class of applications is addressed in a separate paper. Therefore, our focus in this paper is on challenges C1-C5 and P1.

3.1 Loop Preprocessing

To facilitate the execution of loops in this fashion, our compiler first performs an initial preprocessing of the loop. In the process, it may replace an initial loop with a sequence of loops, each of which conforms to a *canonical form*.

Consider any data intensive parallel loop in the dialect of Java described earlier in this paper. For the purpose of our discussion, collections of objects whose elements are modified in the loop are referred to as *left-hand-side* or LHS collections, and the collections whose elements are only read in the loop are considered as *right-hand-side* or RHS collections.

```
foreach(r $\in \mathcal{R}$) {
    $O_1[\mathcal{S}_{L1}(r)] = \mathcal{F}_1(O_1[\mathcal{S}_L(r)], I_1[\mathcal{S}_{R1}(r)], \ldots, I_n[\mathcal{S}_{Rn}(r)])$
    ...
    $O_m[\mathcal{S}_{Lm}(r)] = \mathcal{F}_m(O_1[\mathcal{S}_L(r)], I_1[\mathcal{S}_{R1}(r)], \ldots, I_n[\mathcal{S}_{Rn}(r)])$
}
```

Figure 3: Canonical Form of Loop: Output Collections Fit in Main Memory

The canonical form we support is shown in Figure 2. In the special case when all the output collections can fit in the main memory, we can support a more general form, which is shown in Figure 3.

Initially, consider the canonical loop shown in Figure 2. The domain over which the iterator iterates is denoted by $\mathcal{R}$. Let there be n RHS collection of objects read in this loop, which are denoted by $I_1, \ldots, I_n$. The same RHS collection may be accessed with different subscript functions in the same loop, so these n collections need not be distinct. Similarly, let the LHS collections written in the loop be denoted by $O_1, \ldots, O_m$. All LHS collections are accessed using a single subscript function, $\mathcal{S}_L$. In the iteration r of this loop, the value of output element $O_i[\mathcal{S}_L(r)]$ is updated using the function $\mathcal{F}_i$. This function uses one or more of the values $O_i[\mathcal{S}_L(r)]$, $I_1[\mathcal{S}_R1(r)], \ldots, I_n[\mathcal{S}_R(r)]$, and other scalar values in the program. More specifically, the function has the form:

$$O_i[\mathcal{S}_L(r)] = O_i[\mathcal{S}_L(r)] \; op \; g_1(I_1[\mathcal{S}_{R1}(r)]) \; op \; g_2(I_2[\mathcal{S}_{R2}(r)]) \; op \; \ldots \; op \; g_n(I_n[\mathcal{S}_{Rn}(r)])$$

where, *op* is a associative and commutative operator, and the function g_i only uses $I_i[\mathcal{S}_{Ri}(r)]$ and scalar values in the program.

The two main restrictions we impose in this definition of a canonical loop are:

- All the LHS collections are accessed using the same subscript function. This restriction significantly eases the compilation challenge C1 (tiling) that we listed earlier.
- The updates to the LHS elements are performed using associative and commutative functions only. This restriction significantly eases the compilation challenge C5 (performing the computation associated with each iteration). With this restriction, the elements from different input collections that contribute to the value of an output element need not be brought into the memory at the same time.

Starting from a general data parallel loop, loop fission can be performed to obtain a series of loops, such that each of them conforms to this canonical form [16]. Applying such loop fission may involve introducing temporary collections. However, for the applications we experimented with, loop fission was never required.

In some data intensive applications, the size of the output collections is quite small, and can easily fit in the main memory. In such applications, output collections need not be tiled. Therefore, we can generalize our canonical loop and not require that all LHS collections are accessed using the same subscript function (Figure 3).

3.2 Loop Planning

For each canonical loop, we first perform a runtime preprocessing phase called *loop planning*. This phase is mainly responsible for addressing the challenges C1, C2, C3, and P1, and is carried out using a runtime infrastructure called Active Data Repository (ADR) [9, 10].

P1: The work distribution strategy we use is that each iteration is performed on the owner of the element read in that iteration. As a result, no communication is required for the RHS elements.

The static declarations on the LHS collection can be used to decide the total size of the output required. Not all elements of this LHS space need to be updated on all processors. However, in the absence of any other analysis, we can simply replicate the LHS collections on all processors and perform global reduction after local reductions on all processors have been completed.

C1: For partitioning the output collections into tiles fitting in the main memory, one simple strategy that can be used is as follows. We query the runtime system to determine the available memory that can be allocated on a given processor. Then, we divide the LHS space into blocks of that size. Formally, we divide the LHS range into a set of smaller ranges (called *strips*) $\{S_1, S_2, \ldots, S_r\}$. Since each of the LHS collection of objects in the loop is accessed through the same subscript function, the same strip mining is used for each of them.

For reducing the communication volume, a variation of this strategy is used. On a particular processor, space is allocated for an element only if that element is updated on that processor. More details of this strategy are described in a separate paper that focuses on communication for data intensive applications.

C2: For determining the set of disk blocks that need to be read for performing the updates on a given tile, the compiler extracts an expression that is compared to the *meta-data* associated with each disk block. The details of the compiler technique are described in Section 4.2. For the purpose of describing our execution strategy, we assume that for the RHS collection I_i, on the given processor j, and for the LHS strip l, the set of disk blocks that need to read is denoted by L_{ijl}.

C3: The order in which disk blocks are read is determined entirely by the runtime system, and is transparent to the compiler. The goal is to minimize the seek time, while maximizing the parallelism in accessing the disks, if multiple disks are attached to the same processor [9, 10].

3.3 Loop Execution Strategy

The challenges C4 and C5 are addressed by the compiler as part of the loop execution strategy. We initially describe a simple execution strategy that does not expect any regularity from datasets. We refer to this strategy as the *sparse* execution strategy. Such an execution strategy is ideally suited for applications like the satellite data processing application described in Section 2. An application like multi-grid virtual microscope, described in the appendix, achieves better performance when the regularity in dataset is exploited.

```
For each LHS strip $S_l$:
  Execute on each Processor $P_j$:
    Allocate and initialize strip $S_l$ for $O_1, \ldots, O_m$
    For each RHS collection $I_i$
      For each disk block in $L_{ijl}$
        For each element $e$ in the block
          $\mathcal{I} = Iters(e)$
          For each $i \in \mathcal{I}$
            If $(i \in \mathcal{R}) \wedge (\mathcal{S}_L(i) \in S_l)$
              Update values of $O_1[\mathcal{S}_L(i)], \ldots, O_m[\mathcal{S}_L(i)]$
  Perform global reduction to finalize the values for $S_l$
```

Figure 4: Sparse Loop Execution Strategy

3.3.1 Sparse Execution Strategy

The sparse loop execution strategy is shown in Figure 4. The LHS tiles are allocated one at a time. For each LHS tile S_l, and for each RHS collection I_i, the RHS disk blocks from the set L_{ijl} are read successively. For each element of the disk block in memory, we need to determine the set of iterations in the loop that uses this element. For an element e, the set of iterations is denoted by $Iters(e)$. The method used for computing this set from the source code of the program is described in Section 4.1. For each iteration i belonging to the set $Iters(e)$, we further check if it belongs to the loop range $\mathcal{R}$, and if the element of the LHS collection updated in that loop iteration, $\mathcal{S}_L(i)$, belongs to the tile currently being processed (S_l). After checking for these conditions, we *update* the elements $O_1[\mathcal{S}_L(i)], \ldots, O_m[\mathcal{S}_L(i)]$ using the element e. Note that the requirement that the LHS elements are updated using associative and commutative operators is essential for being able to compute their values without having all the arguments or operands in the main memory at the same time.

In many applications, two simplifications to this general execution strategy are possible. First, all output collections may fit in main memory. In this case, the loop over the LHS strips and the conditional $\mathcal{S}_L(i) \in S_l$ are not required. Because there is no need to tile the output collections, there is no requirement that all output collections must be accessed using the same subscript function (Figure 3).

Second, as we will discuss in Section 4.1, static analysis can determine that any element may be accessed in only a single iteration. In such applications, the loop over all iterations in $Iters(e)$ is obviously not required.

The accumulation function obtained using this strategy for the data parallel code presented in Figure 1 is shown in Figure 5. The procedure in this figure shows the computation performed after reading each disk block. Each element is accessed in only one iteration in this code. Two conditional statements are executed in each iteration. The first conditional checks if the iteration corresponding to this input element belongs to the range of the original loop. The second conditional checks if the output element corresponding to the input element belongs to the current tile. Note that the conditional with invocation of the `getData` function in the Figure 1 is not required in the code shown in Figure 5, because we are only executing iterations that have data elements associated with them.

Though the above execution sequence achieves very high locality in disk accesses, it performs considerably higher computation than the original loop. This is because the mapping from the element e to the set of iterations, inter-

```
void Accumulate(ADR_Box QBox, ADR_Box tile,
     ADR_Box block) {

 ADR_Point opt(2);
 InputValue val;
 val = block.getFirst();
 while (val ! = NULL) {

  {* Intersection with the range *}
  if ((QBox.getHigh()[0] >= args[0]) && (val.x >= args[2]) &&
    (val.y >= args[4] )&& (QBox.getLow()[0] <= args[1]) &&
    (val.x <= args[3]) && (val.y <= args[5])) {

   {* Intersection with the tile *}
   if ((val.x >= tile[0].getLow()[0]) &&
     (val.y >= tile[0].getLow()[1]) &&
     (val.x <= tile[0].getHigh()[0]) &&
     (val.y <= tile[0].getHigh()[1])) {

    {*Accumulation function *}
    opt[0] = val.x ;
    opt[1] = val.y ;
    OutImage[opt].Accumulate(val);
   }
  }
 val = block.getNext();
 }
}
```

Figure 5: Processing Code for Each Disk Block for the Satellite Application

section of an iteration with the loop range, and intersection of the RHS subscript with the current tile needs to be performed while processing each input element.

In this paper, we also propose two mechanisms that can, depending upon the application, potentially reduce the overhead associated with such an execution strategy. In Section 3.3.2, we describe a variation of this execution strategy that can exploit the regularity in the data elements comprising a disk block. In Section 4.3, we describe a technique for hoisting conditional statements which can eliminate most of the additional computation associated with the code shown in Figure 4.

3.3.2 Dense Execution Strategy

In Figure 6, we present a variation of the execution strategy presented in Figure 4. This execution strategy exploits regularity in the set of data elements comprising a disk block to reduce some of the overheads associated with the sparse execution strategy.

The key to exploiting the regularity is in being able to obtain a regular section descriptor of the elements comprising a disk block before processing the individual data elements. If such a regular section descriptor is available, the function $Iters$ can be applied to this descriptor to obtain the set of iterations ($\mathcal{I}$) that use one of the elements in this disk block. The technique for extracting this set from a regular section descriptor is described in Section 4.1. The set of iterations returned by the function are further intersected with the loop range $\mathcal{R}$. For each iteration i belonging to the set $\mathcal{I}$, we need to check if the LHS elements updated are in the current tile, and then update the corresponding LHS elements. If the LHS subscript function $\mathcal{S}_L$ is very simple (e.g. if it is the identity function), the set $\mathcal{I}$ can be intersected with the tile, and the conditional can be avoided. In many other cases, the code motion techniques we describe in Section 4.3 can be used for merging the conditionals with the loop header.

```
For each LHS strip $S_l$:
  Execute on each Processor $P_j$:
    Allocate and initialize strip $S_l$ for $O_1, \ldots, O_m$
    For each RHS collection $I_i$
      For each disk block in $L_{ijl}$
        Extract $\mathcal{D}$, a descriptor of data in the block
        $\mathcal{I} = (Iters(\mathcal{D}) \cap \mathcal{R})$
        For each $i \in \mathcal{I}$
          If $\mathcal{S}_L(i) \in S_l$
            Update values of $O_1[\mathcal{S}_L(i)], \ldots, O_m[\mathcal{S}_L(i)]$
  Perform global reduction to finalize the values for $S_l$
```

Figure 6: Dense Loop Execution Strategy

4 Advanced Compiler Analysis and Optimizations

4.1 Inverting Data Access Functions

In the first execution strategy we presented in Figure 4, a key function used is $Iters(e)$. Given an element e from an input collection, this function determines the iteration(s) of the original loop in which this element is accessed. In the dense execution strategy presented in Figure 6, we further generalized the function $Iters$ to find all iterations in which any element from a regular section of elements may be accessed. In this subsection, we initially present a technique for constructing a function that maps an element in an input collection to a set of iterations. Then, we discuss how this mapping can be extended to work on a regular section of elements.

Consider the satellite data processing template presented in Figure 1, and the multi-grid virtual microscope in Figure 14. The main difficulty in computing the mapping from an element to a set of iterations comes because of the `getData` functions. In the class `SatData` in the satellite data processing template, the function `getData` takes a latitude, longitude, and time, and returns either a pixel or null. In finding such a pixel to return, this function (through a series of other function invocations), looks at the block with the matching time attribute, and sees if that block has any pixel with the specified latitude and longitude values.

In constructing the function $Iters$, we need to take a pixel, and determine the iterations of the data parallel loop in which it will be accessed. This mapping can be considered as a composition of two mappings: 1) mapping from the pixel to argument tuple of the `getData` function, and, 2) mapping from the argument tuple of the `getData` function to the loop iteration. For the satellite data processing example, the second mapping is trivial. In general, we only handle affine mappings from the argument tuple of the `getData` function to the loop iteration. Our main focus is on computing the mapping from an element of the input collection to the argument tuple of the `getData` function.

We compute such a mapping in three main steps:
A. We find the *dominating constraints* for the return statement of the `getData` function, which are the necessary conditions for an element to be returned by an invocation of the `getData` function. For finding such dominating constraints, we examine the control predicates that *dominate* [3] the re-

turn statement(s).

B. Perform symbolic execution [23] to determine if these necessary conditions are *sufficient* also, i.e., if these conditions hold, the element will necessarily be returned. If this step is successful, we have a set of necessary and sufficient conditions, denoted by $\mathcal{C}$, for a particular element to be returned.

C. Construct a mapping from the necessary and sufficient conditions $\mathcal{C}$ to the argument tuple to the `getData` function. We need to find a set of argument tuples to the `getData` function such that 1) if the function is invoked with any of these argument tuples, the conditions $\mathcal{C}$ will necessarily hold, and 2) the conditions $\mathcal{C}$ can only hold if the `getData` function has been invoked with one of the argument tuples in the set computed.

The mapping between the element and the argument tuple of the `getData` function thus computed can be composed with the affine function relating the argument tuple of the `getData` function to the loop iteration to construct the $Iters(e)$ function.

4.1.1 Mapping from a Regular Section of Elements

Next, we consider the case when we need to compute the set of iterations in which any of the elements from a regular section of elements may be accessed. The technique we present for this purpose has one restriction as compared to the technique presented earlier in this subsection. We assume that each element is accessed in a single iteration only.

Let D denote the regular section descriptor of the elements from an input collection. Along the i^{th} dimension, let l_i be the lowest value, s_i be the stride, and h_i be the highest value. Let L denote the coordinate of the lowest point in the section. Our technique involves the following steps:

- Compute, symbolically, the iteration corresponding to an arbitrary element, P. Denote the iteration computed by I.
- For each dimension i of the input collection, compute the iteration corresponding to the element $P + s_i$. Denote the iteration computed by I'_i.
- Compute the difference between I'_i and I, for all dimensions i. This gives the stride s'_i along the i^{th} dimension in the iteration space.
- Compute the iteration corresponding to the lowest element, L. Denote the iteration computed by L'.

The function $Iters(D)$ simply returns a section of iterations defined by the following: 1) the lowest points is L', 2) the stride along the i^{th} dimension is s'_i, and 3) the number of elements along the i^{th} dimension is $\lfloor \frac{h_i - l_i}{s_i} \rfloor$.

The two techniques we have presented in this subsection are obviously not applicable to all codes. However, they turned out to be sufficient for the codes we used in this paper.

4.2 Selecting Disk Blocks for Each Tile

In this subsection, we discuss how the compiler and runtime system addresses the challenge C2 listed in Section 3. Given a tile or strip S_l and a particular LHS collection I_i, we need to determine the set of disk blocks that contribute to the computation of one or more element of S_l. The technique used must include every disk block that can contribute to computation of one or more elements of S_l, but may include some disk blocks which may not contribute to any element of S_l.

The runtime analysis for determining such a set of disk blocks obviously needs to be performed without retrieving all disk blocks. The runtime system maintains an index of all disk blocks corresponding to any LHS collection. Associated with this index is *meta-data* for each disk block, giving a *summary* of the elements comprising the disk block.

In Section 4.1, we discussed a compiler technique for determining the iteration(s) of the loop in which a particular element is accessed. Suppose we have a RHS element e. If i is an iteration in which the element e is accessed, then an element $\mathcal{S}_L \cdot i$ from a LHS collection will be updated using the element e. Therefore, the disk block with the element e must be accessed while processing the tile containing the element $\mathcal{S}_L \cdot i$.

Under the assumption that the LHS subscript function is an affine function, the above reasoning can be used for determining the LHS element(s) that are updated by a particular RHS element. However, the main challenge is that we need to find all disk blocks that need to be accessed for a particular tile, without accessing the disk blocks themselves. We first present the analysis that can be performed for the applications where the dense strategy is applicable, and then extend it to the applications where sparse strategy is used.

For applications where the dense strategy presented in Figure 6 is applicable, a regular section descriptor of the elements comprising a disk block is available. Such a descriptor is stored as part of the *meta-data* associated with each disk block. Let D denote the regular section descriptor. The analysis presented in Section 4.1 can be used to determine I, a regular section descriptor of the iterations of the loop that access elements from the disk block. By applying the LHS subscript function $\mathcal{S}_L$ on the section I, a regular section descriptor of the LHS elements updated can be computed. This section can be intersected with the tile S_l to determine if the disk block needs to be accessed while processing the tile.

For applications where sparse execution strategy is used, this scheme is not directly applicable. This is because a regular section descriptor of the elements comprising a disk block is not available. In such applications, a *bounding box* description of the elements comprising a disk block is stored as part of the meta-data. A bounding box is a rectilinear section, such that the coordinates of all elements in the block fall within that section. Unlike a regular section descriptor, however, not all elements within the bounding box need to be present in the disk block. The rest of the analysis is the same as for the dense applications, except that a bounding box is used instead of the regular section descriptor.

4.3 Code Motion for Conditional Statements

One of the factors causing inefficiency in the code shown in Figure 5 is the cost of evaluating different conditional statements. We now present a technique which eliminates redundant conditional statements and merges the conditionals with loop headers or other conditionals wherever possible.

In Figure 7, we show the *accumulation* function generated by the compiler for the virtual microscope application with subsampling, and without multiple levels of resolution. This is a simple data intensive application which does not require the high-level abstractions we have presented in this paper. This application was used as one of benchmarks in our earlier work [16]. In this code, no conditional is required

```
void Accumulate(ADR_Box block, ADR_Box
  tile, ADR_Box querybox) {
 ADR_Box box = block.intersect(querybox);
 ADR_Pt inputpt(2);
 ADR_Pt outputpt(2);
 int low1 = box.getLow(0);
 int low2 = box.getLow(1);
 int hi1 = box.getHigh(0);
 int hi2 = box.getHigh(1);
 int tlow1 = tile.getLow(0);
 int tlow2 = tile.getLow(1);
 int thi1 = tile.getHigh(0);
 int thi2 = tile.getHigh(1);
 for (i0 = low1; i0 < hi1; i0++) {
  for (i1 = low2; i1 < hi2; i1++) {
   inputpt[0] = i0;
   inputpt[1] = i1;
   outpoint[0] = (i0 - v0) / 2 ;
   outpoint[1] = (i1 - v1) / 2 ;
   if ((tlow1 <= outpoint[0]) &&
         (thi1() >= outpoint[0]) &&
         (tlow2 <= outpoint[1]) &&
         (thi2 >= outpoint[1])) {
    if ((i0 % 2 == 0) && (i1 % 2 == 0)) {
     Output[outputpt].Accum(VScope[inputpt]);
    }
   }
  }
 }
}
```

Figure 7: Processing Code for Each Disk Block for Virtual Microscope with Subsampling

for checking if the element corresponds to an iteration, but a conditional is required to check if the output element updated is within the tile currently being updated. Moreover, there is a conditional in the original data parallel loop. This conditional ensures that only every $subsamp^{th}$ element along each dimension is selected for creating the output image. The value of the subsampling factor (*subsamp*) is 2 in this example. These two conditional can still be folded into the loop header using our proposed technique.

Our technique works in three stages. Initially, for each conditional statement, we compute a term we refer to as *dominating constraints*. The dominating constraints for a conditional statement is the composition of the constraints imposed by all enclosing conditionals and loops. Using the dominating constraints, we can detect redundant conditionals and eliminate them. In the second phase, we consider if the conditionals can be hoisted in the code and can be composed together with other conditionals or loop headers. Finally, we use the omega calculator [22, 34] for looping over the set of iterations that result from composing a loop with one or more conditionals.

Preliminaries: We consider only structured control flow, with `if` statements and `loops`. Within each control level, the *definitions* and *uses* of variables are linked together with *def-use* links. Since we are looking at *def-use* links within a single control level, each use of a variable is linked with at most one definition.

The candidates for code motion in our framework are if statements. One common restriction in code hoisting frameworks like Partial Redundancy Elimination (PRE) [28] and the existing work on code hoisting for conditionals [6, 30] is that syntactically different expressions which may have the same value are considered different candidates. We remove this restriction by following *def-use* links and considering multiple *views* of the expressions in conditionals and loops.

To motivate this, consider two conditional statements, one of which is enclosed in another. Let the outer condition be `x > 2` and let the inner condition be `y > 3`. Syntactically, these are different expressions and therefore, it appears that both of them must be evaluated. However, by seeing the definitions of `x` and `y` that reach these conditional statements, we may be able to relate them. Suppose that `x` is defined as `x = z -3` and `y` is defined as `y = z - 2`. By substituting the definitions of `x` and `y` in the expressions, the conditions become `z - 3 > 2` and `z - 2 > 3`, which are identical.

We define a view of a candidate for code motion as follows. Starting from the conditional, we do a forward substitution of the definition of zero or more variables occurring in the conditional. This process may be repeated if new variables are introduced in the expression after forward substitution is performed. By performing every distinct subset of the set of the all possible forward substitutions, a distinct view of the candidate is obtained. Since we are considering *def-use* within a single control level, there is at most one reaching definition of a use of a variable.

Views of a loop header are created in a similar fashion. Forward substitution is not done for any variable, including the induction variable, which may be modified in the loop.

Phase I: Downward Propagation In the first phase, we propagate *dominating constraints* down the levels, and eliminate any conditional which may be redundant. Consider any loop header or conditional statement. The range of the loop or the condition imposes a constraint for values of variables or expression in the control blocks enclosed within. As described previously, we compute the different views of the constraints by performing a different set of forward substitutions. By composing the different views of the loop headers and conditionals statements, we get different views of the *dominating constraints*.

Consider any conditional statement for which the different views of the dominating constraints are available. By comparing the different views of this conditional with the different views of dominating constraints, we determine if this conditional is redundant. A redundant conditional is simple removed and the statements enclosed inside it are merged with the control block in which the conditional statement was initially placed.

Phase II: Upward Propagation After the redundant conditionals have been eliminated, we consider if any of the conditionals can be folded into any of the conditionals or loops enclosing it. We initially compute all the views of the conditional which is the candidate for hoisting.

Consider any statement which dominates the conditional. We compute two terms: *anticipability* of the candidate at that statement, and *anticipable views*. The candidate is anticipable at its original location and all views of the candidate computed originally are anticipable views.

The candidate is considered anticipable at the beginning of a statement if it is anticipable at the end of the statement and any assignment made in the statement is not live at the

```
R= {[i,j] : low1 <= i <= hi1
     and low2 <= j <= hi2
     and exists (ii : ii * 2 = i)
     and exists (jj : jj * 2 = j)
   } ;

S:= {[i,j] : tlow1 * 2 + v0 <= i <= thi1 * 2 + v0
     and tlow2 * 2 + v1 <= j <= thi2 * 2 + v1
   } ;

U:= (R intersection S);
codegen U;
```

```
if (2 * low2 <= -v1 + thi2 && low2 <= v1 + 2 * thi2) {
 for (t1 = max(2 * (v0 + 2 * tlow1 + 1)/2, 2 * (low1 + 1)/2);
        t1 <= min(v0 + 2 * thi1, hi1) ; t1 += 2) {
  for (t2 = max(2 * (2 * tlow2 + v1 + 1)/2, 2 * (low2 + 1)/2);
        t2 <= min(v1 + 2 * thi2, hi2) ; t2 += 2) {
   s1(t1, t2);
  }
 }
}
```

Figure 8: Using Omega Calculator for Code Generation: Input to the Calculator (left) and Calculator's Output (right)

end of the conditional. This motivation behind this definition is as follows. A statement can only be folded inside the conditional only if the values computed in it are used inside the conditional only. To compute the set of anticipable views at the beginning of a statement, we consider two cases:
Case 1: If the variable assigned in the statement does not influence the expression inside the conditional, all the views anticipable at the end of the statement are anticipable at the beginning of the statement.
Case 2: Otherwise, let the variable assigned in this statement be v. From the set of views anticipable at the end of the statement, we exclude the views in which the definition of v at this statement is not forward substituted.

Phase III: Code Generation Consider any conditional or loop which encloses the original candidate for placement, and let this candidate be anticipable at the beginning of the first statement enclosed in the conditional or loop. We compare all the views of this conditional or loop against all anticipable views of the candidate for placement. If either the left-hand-side or the right-hand-side of the expression are identical or separated by a constant, we can compose the candidate into this conditional or loop.

We use the integer set manipulation ability of omega calculator [22, 34] to compose multiple conditionals and loops. In Figure 7, the two conditionals can be merged together with the loop header. The input and output from omega calculator for generating such a code is shown in Figure 8.

5 Experimental Results

In this section we present experimental results to demonstrate the effectiveness of our compilation techniques. A prototype compiler implementing our techniques has been developed using the Titanium infrastructure from Berkeley [38]. We used a cluster of 400 MHz Pentium II based computers connected by a gigabit switch. Each node has 256 MB of main memory and 18 GB of local disk. We ran experiments using 1, 2, 4 and 8 nodes of the cluster.

We experimented with three different applications. The first is a satellite image processing application, similar to the code in Figure 1, and referred to as `satellite` in this section. The processing for this application was described in Section 2.1. One difference in the version we implemented is that we further reduce the output size by collapsing every 2 × 2 block of the output into a single value. The other two applications are based on the Virtual Microscope [14]. We present results from the multi-grid version of virtual microscope, which is described in the appendix. This application is referred to as `mg-vscope` in this section. Finally, we present experimental results from a regular version of virtual microscope, which is referred to as `vscope`. A set of simple compilation techniques, described in our earlier work [16], are sufficient for generating correct code for the `vscope` application. Results from `vscope` are included here to demonstrate benefits from the conditional hoisting technique we have presented.

We created three version each for the the first two applications, `satellite` and `mg-vscope`. The first version uses the execution model as described in Section 3.3.1 and is referred to as `sparse`. The second version, referred to as `dense`, uses the execution model as described in Section 3.3.2. The last version of each application was created by applying the conditional hoisting optimization, as described in Section 4.3, to the version that performed the best among the first two. We refer to this version as `opt`. We executed each of these versions with two different *query sizes*. By query size, we imply the size of the input dataset the application processes during execution. We refer to the two query sizes used as `large` and `medium`, respectively.

The entire dataset for the `satellite` application contains data for the entire earth at a resolution of $1/128^{th}$ of a degree in latitude and longitude, and 15,000 time steps, corresponding to a total of 2.7GB of data. The `large` query size corresponds to a region of $15,000 \times 20,000 \times 20,000$ points. It involves reading 1.9GB of data and producing an output of 400MB. The `medium` query size corresponds to traversing a region of $15,000 \times 10,000 \times 10,000$ points, requiring the application to read 446MB and produce an output of 100MB.

The entire dataset for `mg-vscope` application corresponds to an image of $29,328 \times 28,800$ pixels collected at five different magnification levels, with a total size of 3.3GB. The `large` query processes a region of $20,000 \times 20,000$ pixels, requiring about 3GB of data to be read and an output of 1.6GB to be produced. The `medium` query processes a region of $10,000 \times 10,000$ pixels which requires reading around 627MB. The output in this case is 400MB.

Figure 9 compares the different versions of the satellite application with the `large` query. Our experiments show that the `sparse` version runs faster by a factor of 5 to 10 than the `dense` version. This is because of the nature of the dataset for this application. It is irregular, which causes the disk blocks to contain data for large areas. The data is also sparse, so for a large portion of the area covered by each block of data, only a small portion of the points within that area actually contains data. The `opt` version

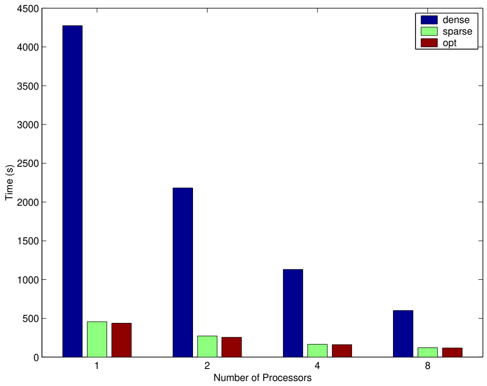

Figure 9: Comparing `sparse`, `dense`, and `opt` versions of `satellite` with `large` query.

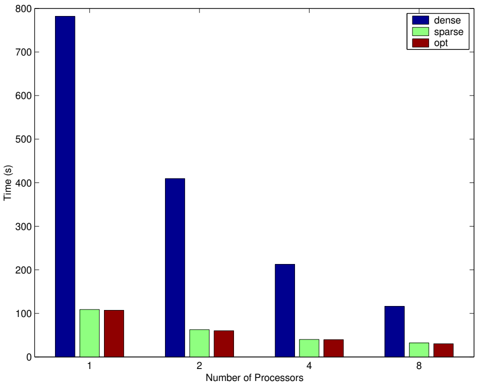

Figure 10: Comparing `sparse`, `dense`, and `opt` Versions of `satellite` with `medium` Query

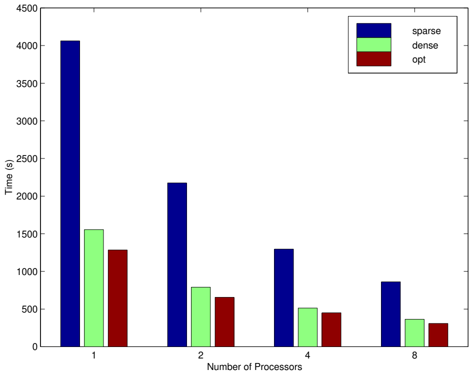

Figure 11: Comparing `sparse`, `dense`, and `opt` Versions of `mg-vscope` with `large` Query

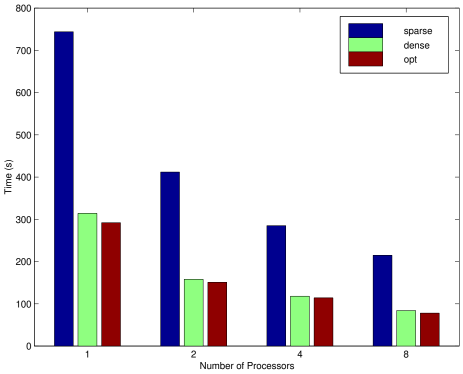

Figure 12: Comparing `sparse`, `dense`, and `opt` Versions of `mg-vscope` with `medium` Query

corresponds applying the conditional hoisting optimization to the `sparse` version. For this application, the optimization is able to eliminate one out of the two conditionals inside the innermost loop. This resulted in improvements of the execution time of 3.96%, 6.25%, 2.22%, and 2.99%, on 1, 2, 4 and 8 processors, respectively. The speedups of the final version are 1.71, 2.72, and 3.71, on 2, 4, and 8 processors, respectively.

Figure 10 shows the results for the `medium` query The `sparse` version is 4 to 8 times faster than `dense` version. The conditional motion optimization further improved the `sparse` version by 1.63%, 3.78%, 0.94% and 5.76% on 1, 2, 4 and 8 processors respectively. The speedups of the final version are 1.78, 2.69, and 3.3, on 2, 4, and 8 processors, respectively.

The results from the `mg-vscope` application with the `large` query are summarized in Figure 11. The experiments show that the `dense` version is 2 to 3 times faster than the `sparse` version. This is because the dense version does not involve any redundant iterations, and eliminates the overheads associated with each iteration. We then applied the conditional hoisting optimizations to the `dense` version. This resulted in improvements of 17.36%, 16.90%, 12.31% and 14.98% on 1, 2, 4 and 8 nodes, respectively. For this application, the conditional hoisting optimization achieves more significant gains, because besides eliminating one of the conditionals, it also merges a conditional with the bounds of the loop. The speedups of the final version are 1.96, 2.85, and 4.15, on 2, 4, and 8 processors, respectively.

The results for the `medium` query are shown in Figure 12. For this query size, the `dense` version is again faster by nearly 2 to 3 times than the `sparse` version. Conditional hoisting optimization yields extra 7.11%, 4.80%, 3.13% and 7.15% improvement on 1, 2, 4 and 8 nodes, respectively. The speedups of the final version are 1.93, 2.56, and 3.75, on 2, 4, and 8 processors, respectively.

Finally, we describe the results from the `vscope` application. It will traverse a regular dataset, and generates an image at a user specified (lower) resolution. It accomplishes this by subsampling the original data. The dataset for this application is a $19,700 \times 15,360$ pixel image from a microscope. The application is generating an image that corresponds to an area of $10,000 \times 10,000$ of that image, reduced by a factor of 8 on each dimension.

The execution times for the compiler generated versions without the conditional hoisting optimization and with the conditional hoisting optimization are presented in Figure 13. The improvement in performance measured on the experiments were 23.23%, 27.88%, 29.38%, and 35.77% on 1, 2, 4, and 8 processors, respectively. We notice that conditional motion for this application has a substantially higher impact than on the other two applications. This is because

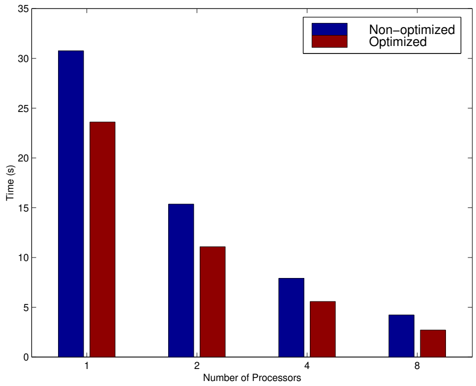

Figure 13: `vscope` with and without conditional hoisting

this application involves a conditional as part of its reduction function, besides the conditionals introduced by the compiler. Only every 8^{th} element along each dimension is actually read and processed. Our conditional hoisting technique is successful in merging the conditional from the reduction function with the inner-loop, resulting in significant performance improvements.

6 Related Work

The work presented in this paper is part of our continuing work on compiling data intensive applications [16, 15]. Our initial work focused on regular data intensive applications, where high-level abstractions were not required, and the compiler analysis was much simpler [16]. The basic ideas behind the sparse execution strategy were presented in a previous workshop paper [15]. This paper makes several new contributions as compared to our previously published work. These contributions include 1) the formulation of high-level abstractions, 2) more rigorous and complete presentation of the execution strategies, 3) compiler techniques for supporting the execution strategy (Sections 4.1 and 4.2), and 4) experimental results based upon an actual compiler implementation.

Our code execution model has several similarities to the *data-centric* locality transformations proposed by Pingali *et al.* [24]. We fetch a data-chunk or shackle from a lower level in the memory hierarchy and perform the iterations from the loop which use elements from this data-chunk. We have focused on applications where no computations may be performed as part of many iterations from the original loop. So, instead of following the same loop pattern and inserting conditionals to see if the data accessed in the iteration belongs to the current data-chunk, we compute a mapping function from elements to iterations and iterate over the data elements.

Our work on supporting high-level abstractions has been motivated, in part, by recent work of Pingali *et al.* on applying generic programming to sparse computations [27]. However, since we focus on a different class of applications, and on disk-resident datasets, the details of the work are very different. Our work is also somewhat similar in flavor to the efforts on synthesizing sparse codes from dense ones [4, 26, 35].

Several other researchers have focused on removing fully or partially redundant conditionals from code. Mueller and Whalley have proposed analysis within a single loop-nest [30] and Bodik, Gupta, and Soffa perform demand-driven interprocedural analysis [6]. Our method is more aggressive in the sense we associate the definitions of variables involved in the conditionals and loop headers. This allows us to consider conditionals that are syntactically different. Our method is also more restricted than these previously proposed approaches in the sense that we do not consider partially redundant conditionals and do not restructure the control flow to eliminate more conditionals. Many other researchers have presented techniques to detect the equality or implies relationship between conditionals, which are powerful enough to take care of syntactic differences between expressions [5, 17, 37]. Integer set manipulation has also been previously used for many specific optimization and code generation problems in parallel compilation [1, 19, 36].

Our work on providing high-level support for data intensive computing can be considered as developing an out-of-core Java compiler. Compiler optimizations for improving I/O accesses have been considered by several projects [8, 21, 20, 32, 29]. These projects have concentrated on relatively simple stencil computations, with regular datasets. Our work is significantly different in the class of applications we focus on, and in our support for high-level abstractions of sparse datasets.

7 Conclusions

High-level abstractions for accessing datasets can significantly simplify development of applications. With the help of aggressive compilation techniques, such high-level abstractions can be supported without compromising performance. In this paper, we have presented a set of compiler techniques to support high-level abstractions for complex and/or irregular disk-resident datasets. Our set of techniques include a *sparse* execution strategy, a *dense* execution strategy, two static analyses for supporting these execution strategies, and a technique for improving performing by hoisting of conditionals.

We have developed a prototype compiler implementing these techniques. Our results so far have been very promising. The main observations from our current implementation and experiments include:

- The choice of execution strategies is critical for performance. For `satellite`, where the sparse execution strategy was preferable, nearly a factor of 10 difference in performance is seen between sparse and dense versions. For `mg-vscope`, where the dense execution strategy is preferable, nearly a factor of 2 difference is seen between dense and sparse versions.

- The static analysis techniques for inverting data access functions and selecting disk blocks for each tile are critical for correct and efficient code generation using our execution strategies. Though the techniques we have presented in this paper have several limitations, they turned out to be sufficient for the codes we experimented with.

- Our techniques for hoisting conditionals resulted in up to 30% improvement in performance.

References

[1] Vikram Adve and John Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 186–198. ACM Press, June 1998. ACM SIGPLAN Notices, Vol. 33, No. 5.

[2] Asmara Afework, Michael D. Beynon, Fabian Bustamante, Angelo Demarzo, Renato Ferreira, Robert Miller, Mark Silberman, Joel Saltz, Alan Sussman,

and Hubert Tsang. Digital dynamic telepathology - the Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*. American Medical Informatics Association, November 1998.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] Aart Bik and Harry A. G. Wijshoff. Compilation techniques for sparse matrix computations. In *Proceedings of International Conference on Supercomputing (ICS)*, pages 416–424. ACM Press, July 1993.

[5] W. Blume and R. Eigenmann. Demand-driven, symbolic range propagation. *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, pages 141–160, August 1995.

[6] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Interprocedural conditional branch elimination. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 146–158. ACM Press, June 1997.

[7] Francois Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3), Fall 1993.

[8] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 1–10. ACM Press, July 1995. ACM SIGPLAN Notices, Vol. 30, No. 8.

[9] C. Chang, A. Acharya, A. Sussman, and J. Saltz. T2: A customizable parallel database for multi-dimensional data. *ACM SIGMOD Record*, 27(1):58–66, March 1998.

[10] Chialin Chang, Renato Ferreira, Alan Sussman, and Joel Saltz. Infrastructure for building parallel database systems for multi-dimensional data. In *Proceedings of the Second Merged IPPS/SPDP (13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing)*. IEEE Computer Society Press, April 1999.

[11] Chialin Chang, Bongki Moon, Anurag Acharya, Carter Shock, Alan Sussman, and Joel Saltz. Titan: A high performance remote-sensing database. In *Proceedings of the 1997 International Conference on Data Engineering*, pages 375–384. IEEE Computer Society Press, April 1997.

[12] Chialin Chang, Alan Sussman, and Joel Saltz. Scheduling in a high performance remote-sensing data server. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1997.

[13] A.A. Chien and W.J. Dally. Concurrent aggregates (CA). In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 187–196. ACM Press, March 1990.

[14] R. Ferreira, B. Moon, J. Humphries, A. Sussman, J. Saltz, R. Miller, and A. Demarzo. The Virtual Microscope. In *Proceedings of the 1997 AMIA Annual Fall Symposium*, pages 449–453. American Medical Informatics Association, Hanley and Belfus, Inc., October 1997. Also available as University of Maryland Technical Report CS-TR-3777 and UMIACS-TR-97-35.

[15] Renato Ferreira, Gagan Agrawal, Ruoming Jin, and Joel Saltz. Compiling data intensive applications with spatial coordinates. In *Proceedings of Languages and Compiler for Parallel Computing*, August 2000.

[16] Renato Ferreira, Gagan Agrawal, and Joel Saltz. Compiling object-oriented data intensive computations. In *Proceedings of the 2000 International Conference on Supercomputing*, May 2000.

[17] M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. In *Proceedings of Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1999.

[18] High Performance Fortran Forum. Hpf language specification, version 2.0. Available from http://www.crpc.rice.edu/HPFF/versions/hpf2/files/hpf-v20.ps.gz, January 1997.

[19] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujan, and N. Shenoy. A global communication optimization technique based on data-flow analysis and linear algebra. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(6):1251–1297, November 1999.

[20] M. Kandemir, A. Choudhary, J. Ramanujam, and M. A.. Kandaswamy. A unified framework for optimizing locality, parallelism, and comunication in out-of-core computations. *IEEE Transactions on Parallel and Distributed Systems*, 11(9):648–662, 2000.

[21] M. Kandemir, J. Ramanujam, and A. Choudhary. Improving the performance of out-of-core computations. In *Proceedings of International Conference on Parallel Processing*, August 1997.

[22] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. The Omega calculator and library, version 1.1.0. November 1996.

[23] R. Kemmerer and S. Eckmann. UNISEX: A UNIx-based Symbolic EXecutor for pascal. *Software Practice and Experience*, 15(5):439–458, May 1985.

[24] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 346–357, June 1997.

[25] Tahsin M. Kurc, Alan Sussman, and Joel Saltz. Coupling multiple simulations via a high performance customizable database system. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1999.

[26] Vladimir Lotlyar, Keshav Pingali, and Paul Stodghill. A relational approach to the compilation of sparse matrix programs. In *Proceedings of EUROPAR-97*, August 1997.

[27] Nikolay Mateev, Keshav Pingali, Paul Stodghill, and Vladimir Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *Proceedings of International Conference on Supercomputing (ICS), 2000*, pages 88–100. ACM Press, May 2000.

[28] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[29] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *Proceedings of the Second Symposium on Operating Systems Design and plementation (OSDI '96)*, Nov 1996.

[30] Frank Mueller and David B. Whalley. Avoiding conditional branches by code replication. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 56–66, La Jolla, California, 18–21 June 1995. *SIGPLAN Notices* 30(6), June 1995.

[31] NASA Goddard Distributed Active Archive Center (DAAC). Advanced Very High Resolution Radiometer Global Area Coverage (AVHRR GAC) data. *http://daac.gsfc.nasa.gov/CAMPAIGN_DOCS/ LAND_BIO/origins.html.*

[32] M. Paleczny, K. Kennedy, and C. Koelbel. Compiler support for out-of-core arrays on parallel machines. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 110–118. IEEE Computer Society Press, February 1995.

[33] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '94)*, pages 324–340, October 1994.

[34] William Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[35] William Pugh and Tatiana Shpeisman. Generation of efficient code for sparse matrix computations. In *Proceedings of the Eleventh Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science (LNCS)*, August 1998.

[36] William Pugh and David Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Transactions on Programming Languages and Systems*, 16(4):1248–1278, July 1994.

[37] Peng Tu and David Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 414–423, 1995.

[38] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Libit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. *Concurrency Practice and Experience*, 9(11), November 1998.

Appendix: Multi-grid Virtual Microscope Example

We now describe the multi-grid virtual microscope applications, which is another data intensive applications with sparse datasets. The data parallel code implementing this application is shown in Figure 14.

Application Description

The Virtual Microscope [14] is an application to support the need to interactively view and process digitized data arising from tissue specimens. The raw data for such a system is captured by digitally scanning collections of full microscope slides under high power. In a typical dataset available when a virtual microscope is used in a distributed setting, the same portion of a slide may be available at different resolution levels, but the entire slide is not available at all resolution levels.

A particular user is interested in viewing a rectangular section of the image at a specified resolution level. In computing each component of this rectangular section (output), it is first examined if the that portion is already available at the specified resolution. If it is not available, then we next examine if it is available at a higher resolution (i.e., at a smaller granularity). If so, the output portion is computed by averaging the pixels of the image at the next higher level of granularity. If it is only available at a lower resolution, then the pixels from the lower resolution image are expanded to create the output.

For simplicity, the code presented in Figure 14 assumes that the user is only interested in viewing the image at the highest possible resolution level, which means that subsampling is never done to produce the output image.

```
Interface Reducinterface {
   {* Any object of any class implementing *}
   {* this interface is a reduction variable *}
}
public class VMPixel {
   byte color[3];
}
public class VMPixelOut implements Reducinterface {
   byte color[3];
   int scale;
   void Accumulate(VMPixel el, int sc) {
      if (scale > sc) {
         scale = sc;
         color[0] = el.color[0];
         color[1] = el.color[1];
         color[2] = el.color[2];
      }
   }
}
class ImageBlock {
   VMPixel[2d] BlockData;
   VMPixel getData(Point[2] p) {
      if (BlockData.Intersect(p))
         return BlockData[p];
      else
         return NULL;
   }
}
class ListOfBlocks {
   {* Implements a list of ImageBlock *}
}
```

```
public class InputDataType {
   ListOfBlocks[NUMRES] ResolutionArray;
   VMPixel getData(Point[3] p) {
      ImageBlock blk = ResolutionArray[p[0]].getFirst();
      VMPixel el = NULL;
      while (blk != NULL) {
         q[0] = p[1] / p[0];
         q[1] = p[2] / p[0];
         el = blk.getData(q);
         if (el != NULL)
            return el;
         blk = ResolutionArray[p[0]].getNext();
      }
      return NULL;
   }
}
public class MultiResVScope {
   static int main(int[] args) {
      InputDataType InputData;
      Point[3] abslo = (MIN_SCALE, args[0], args[1]);
      Point[3] abshi = (MAX_SCALE, args[2], args[3]);
      RectDomain[3] AbstractRange = [abslo : abshi];
      Point[2] outlo = (args[0], args[1]);
      Point[2] outhi = (args[2], args[3]);
      RectDomain[2] OutputRange = [outlo : outhi];
      VMPixelOut[2d] OutputData = new
               VMPixelOut[OutputRange];
      Point[3] p;
      Point[2] q;
      VMPixel el;
      foreach (p in AbstractRange) {
         if ((el = InputData.getData(p)) != NULL) {
            q[0] = p[1];
            q[1] = p[2];
            OutputData[q].Accumulate(el, p[0]);
         }
      }
   }
}
```

Figure 14: Multigrid Virtual Microscope Example

Low Level Data Layout

The image is stored as a set of members of class `ImageBlock`. Each `ImageBlock` comprises one rectangular portion of the image stored at a particular resolution level. All of the blocks at a particular resolution level are linked together using the class `ListOfBlocks`.

The classes `ImageBlock` and `ListOfBlocks` are not visible to the programmer writing the processing code.

High Level Abstractions

The class `InputDataType` and its function `getData` give the impression to the programmer that the data is available as a sparse three dimensional array. The resolution level is one of the dimensions of this array. Pixels for possibly more than one, but not all resolution levels are available for any point in the slide.

The main function takes 6 arguments, denoting the two end points of a three dimensional array section. The output computed is two dimensional.