Enabling Code Generation within the Sparse Polyhedral Framework

Alan LaMielle, Michelle Strout
Colorado State University
{lamielle,mstrout}@cs.colostate.edu

March 16, 2010

Technical Report CS-10-102

Enabling Code Generation within the Sparse Polyhedral Framework

Alan LaMielle, Michelle Strout
Colorado State University
{lamielle,mstrout}@cs.colostate.edu

March 16, 2010

Abstract

Loop transformation frameworks based on the polyhedral model use libraries such as Polylib, ISL, and Omega to represent and manipulate polyhedra and use tools like CLooG to generate loops that scan the modified polyhedra. Most of these libraries are restricted to iteration space sets and memory/array access functions with affine constraints that preclude the specification of run-time reordering transformations (i.e., inspector/executor strategies) within the existing code generation tools. Automatic generation of inspector and executor code is important for the parallelization and data locality improvements in irregular computations such as those that manipulate sparse data structures. We enable the specification of run-time reordering transformations at compile time in the Sparse Polyhedral Framework (SPF) by representing indirect memory references and run-time generated data and iteration reorderings using uninterpreted function symbols. This paper presents techniques for manipulating abstract sets and relations that include affine constraints with uninterpreted function symbols thus enabling code generation for run-time reordering transformations in the SPF.

1 Introduction

Applications such as molecular dynamics simulations and finite element analysis that manipulate sparse data structures have performance problems due to indirect memory accesses such as `A[B[i]]`. A number of inspector/executor strategies [13] have been developed to improve the data locality [9] and to parallelize [20] such applications. To automate the application of inspector/executor strategies individually and in composition, we are developing an inspector/executor code generator.

We base the generation of inspector and executor code on the specification of irregular computations and run-time reordering transformations in a previously presented framework [19] that we now call the Sparse Polyhedral Framework (SPF[1]). In [19], we introduced the concept of representing run-time reordering transformations (RTRTs) as Presburger relations with uninterpreted function symbols. Conceptually, an uninterpreted function symbol $f(p_1, p_2, ..., p_3)$ is a function whose output value is not known. Therefore, we use uninterpreted function symbols to represent index arrays, permutation reordering functions, and grouping/tiling functions at compile time even though their values will not be known until runtime. Mathematically, this approach is powerful enough to represent and transform irregular applications.

Current loop transformation frameworks such as Pluto [7] represent and manipulate iteration spaces as polyhedra and/or unions of polyhedra. Code is generated to scan the transformed iteration spaces by (1) determining loop bounds that are affine functions of outer iterators and symbolic constants and (2) computing new memory/array accesses in

[1]This term was originally coined by Larry Carter.

```
for(i=0; i<=5; i++) {
  for(j=0; j<=4; j++) {
    A[i,j]=f(...,A[i-1,j+1],...);
  }
}
```

Figure 1: Original affine computation

```
for(i'=0; i'<=9; i'++) {
  for(j'=max(i',0);
      j'<=min(i',5); j'++) {
    A[j',i'-j']=
      f(...,A[j'-1,i'-j'+1]+1,...);
  }
}
```

Figure 2: Transformed example affine computation

terms of the new iterators. Fourier-Motzkin elimination is used to project out existentially quantified variables and to project out inner iterators in succession to determine the loop bounds for outer iterators.

As an example of what polyhedral transformation frameworks can do, consider the original and transformed affine computation in Figures 1 and 2. The iteration space specification for the original 2D loop is the set

$$\{[\mathrm{i},\mathrm{j}] : (0 \le \mathrm{i} \le 5) \wedge (0 \le \mathrm{j} \le 4)\}.$$

After applying a skewing transformation and a loop permutation transformation, the set specification for the transformed iteration space is

$$\{[\mathrm{i}',\mathrm{j}'] : (0 \le \mathrm{i} \le 5) \wedge (0 \le \mathrm{j} \le 4) \wedge (\mathrm{i}' = \mathrm{i} + \mathrm{j}) \wedge (\mathrm{j}' = \mathrm{i})\}.$$

Generating code for the transformed computation requires determining loop bounds for the new loop iterators (i.e., `i'` and `j'`) and determining the new array access functions within the context of the transformed iteration space (i.e., `A[i,j]` to `A[j',i'-j']`). Fourier-Motzkin elimination is a common technique used to project out the existentials `i` and `j` from the transformed iteration space and modified access function. Fourier-Motzkin elimination is also used to project out `j'` to derive the loop bounds for `i'`.

Now consider the original and transformed irregular computation in Figures 3 and 4. The original iteration space specification for the single "statement" is

$$\mathrm{I} = \{[\mathrm{time},\mathrm{tri}] : (0 \le \mathrm{time} < \mathrm{T}) \wedge (0 \le \mathrm{tri} < \mathrm{R})\}.$$

The iteration space set specification after applying data and iteration reorderings and a partitioning is

$$\mathrm{I}' = \{[\mathrm{time},\mathrm{p},\mathrm{tri}] : (0 \le \mathrm{time} < \mathrm{T}) \wedge \mathrm{p} = \mathrm{proc}(\mathrm{tri}) \wedge (0 \le \mathrm{tri} < \mathrm{R})\}.$$

```
for(time=0; time<T; time++) {
  ...
   for(int tri=0; tri<R; tri++) {
     ...
  ...data[n1[tri]]...
     ...
   }
  ...
}
```

Figure 3: Original irregular computation

```
for(time=0; time<T; time++) {
  ...
  for(p=0; p<NUM_PROCS; p++) {
    for(tri=0; tri<R; tri++) {
      if(p==proc(tri)) {
        ...
        ...data[sigma[n1[delta^-1[tri]]]]...
        ...
      }
    }
  }
  ...
}
```

Figure 4: Transformed irregular computation

Note that there are no affine bounds for the iterator `p`. The original access function for the access to the array `data[]` is

$$\mathrm{A_{orig}} = \{[\mathrm{time}, \mathrm{tri}] \rightarrow [\mathrm{out}] : \mathrm{out} = \mathrm{n1}(\mathrm{tri})\}.$$

Note the use of the uninterpreted function symbol (UFS) `n1` to represent an index array of the same name. The access function for the access to the array `data[]` after the transformation is

$$\mathrm{A'_{orig}} = \{[\mathrm{time}, \mathrm{p}, \mathrm{k}] \rightarrow [\mathrm{out}] : \mathrm{k} = \mathrm{delta}(\mathrm{tri}) \wedge \mathrm{out} = \mathrm{sigma}(\mathrm{n1}(\mathrm{tri}))\}.$$

Note that, following transformation, this specification contains two constraints with the existential `tri` that is not in the final code and that both of those constraints contain uninterpreted function symbols (UFSs).

The problem is that existing libraries and techniques such as FM do not support projecting variables out of constraints involving uninterpreted function symbols. This paper discusses techniques to project out such existentials and introduce affine bounds to enable code generation for inspectors and executors (Section 3). Before introducing these techniques, in Section 2 we discuss the mathematical framework this work is based on. Section 4 evaluates these techniques by comparing our implementation's performance and ability to project out existentials to Omega, a similar library. Finally, we discuss related work in Section 5 and conclude in Section 6.

2 The Sparse Polyhedral Framework (SPF)

We originally introduced the sparse polyhedral framework in [19] where it was described as a compile-time framework for composing run-time reordering transformations. In this section, we provide a basic introduction to the SPF: how to represent computations in the SPF, how to transform these computations, and introduce a new problem that arises in this context. The SPF provides a mathematical framework for representing and transforming irregular computations in a way that is analogous to the polyhedral model for regular computations. SPF builds on the work discussed by Kelly and Pugh in [11], which is based on the concept of Presburger sets and relations. The Kelly and Pugh framework has the ability to express computations and transformations in the polyhedral model as well non-affine memory references and control flow using uninterpreted function symbols (UFSs) [18]. The key addition to the SPF over the Kelly and Pugh framework is the usage of UFSs to express run-time entities such as index arrays and run-time reordering transformations at compile-time and the requirement that the input and output domains of UFSs be specified to enable code generation. We now review the concept of sets and relations, a few operations on these sets and relations, and finish with a discussion of the problems that inhibit code generation.

2.1 Sets

We use sets to represent computation spaces. A set represents an unordered collection of integer tuples in $\mathbb{Z}^m$. For the example in Figure 3, the computation space is specified with the set

$$\mathrm{I} = \{[\mathrm{time}, \mathrm{tri}] : (0 \leq \mathrm{time} < \mathrm{T}) \wedge (0 \leq \mathrm{tri} < \mathrm{R})\}.$$

In general sets have the form

$$s = \{[x_1, \ldots, x_m] : c_1 \wedge \ldots \wedge c_n\}, \tag{1}$$

where each x_i is a tuple variable/iterator and each c_j is a constraint. The constraints in a set are equalities and inequalities that are affine expressions involving the tuple variables, symbolic constants, and existentials. A symbolic constant represents a constant value that does not change during the computation, but may not be known until runtime. An existential is any variable in the constraints that is not a tuple variable or a symbolic constant and represents an existentially quantified variable. Our definition of sets supports a single level of existential quantification and no support for universal quantifiers. The above set s is said to have *arity* m as it has m tuple variables.

Sets can also be unions of collections of integer tuples and have the form

$$s = \{\vec{x} : C_1\} \vee \{\vec{x} : C_2\} \vee \ldots \vee \{\vec{x} : C_p\}, \tag{2}$$

where $\vec{x}$ represents the vector of tuple variables and each C_i represents the constraints for each component of the union. We refer to the complete union of multiple sets as a disjunction and each individual component of the union as a conjunction. This arises from the fact that the constraints of a set are in disjunctive normal form.

Sets in the SPF can also have constraints with uninterpreted function symbol (UFS) expressions. The following iteration space for the example computation in Figure 4 contains the UFS `proc`:

$$\mathrm{I}' = \{[\mathrm{time}, \mathrm{p}, \mathrm{tri}] : (0 \leq \mathrm{time} < \mathrm{T}) \wedge \mathrm{p} = \mathrm{proc}(\mathrm{tri}) \wedge (0 \leq \mathrm{tri} < \mathrm{R})\}.$$

An uninterpreted function symbol has the form $f(a_1, a_2, \ldots, a_m)$, where f is the name of the function and a_1 through a_m are the m arguments to the function. Just as with constraints, the arguments are affine expressions involving the tuple variables, symbolic constants, existentials, and other uninterpreted function symbols (allowing for the possibility of nested functions). Since a UFS is a function, it holds that for a UFS f, if $i = j$ then $f(i) = f(j)$. We use UFSs to represent entities in irregular computations that will not be known until runtime, such as index arrays (`n1`) and run-time reorderings (`proc`). The domain and range of a UFS must be a union of polyhedra and thus cannot be expressed in terms of other UFSs. Also, the arguments to UFS must be UFS expressions (i.e., affine and UFS) of tuple variables and symbolic constants. An existential can only be an argument to a UFS that is a bijection with only one argument.

2.2 Relations

We use relations to represent memory access functions, scheduling functions, and transformation functions. A relation represents an unordered mapping of integer tuples from $\mathbb{Z}^m$ to $\mathbb{Z}^n$. For the example in Figure 3, the access to the data array `data[]` is defined by the relation

$$\mathrm{A}_{\mathrm{orig}} = \{[\mathrm{time}, \mathrm{tri}] \rightarrow [\mathrm{k}] : \mathrm{k} = \mathrm{n1}(\mathrm{tri})\}.$$

This access function defines the indices of the data array `data[]` that are accessed given the loop iterators `time` and `tri`. Notice here the use of the UFS `n1`, an example of using a UFS to represent an *index array*. Index arrays introduce a layer of indirection when accessing a data array and thus only at runtime can we know exactly what elements of the data array are accessed.

In general relations have the form

$$r = \{[x_1, \ldots, x_m] \rightarrow [y_1, \ldots, y_n] : c_1 \wedge \ldots \wedge c_p\}, \tag{3}$$

where each x_i is an input tuple variable, each y_j is an output tuple variable, and each c_k is a constraint. The constraints of a relation follow the same restrictions as set constraints. The above relation r is said to have an *input arity* of m and an *output arity* of n as it has m input tuple variables and n output tuple variables.

Unions of relations are also possible and have the following form (similar to unions of sets)

$$\{\vec{x} \rightarrow \vec{y} : C_1\} \vee \{\vec{x} \rightarrow \vec{y} : C_2\} \vee \ldots \vee \{\vec{x} \rightarrow \vec{y} : C_q\}, \tag{4}$$

where $\vec{x}$ is the vector of input tuple variables, $\vec{y}$ is the vector of output tuple variables, and each C_i represents the constraints for each component in the union.

As a second example, the scheduling/scattering [5, 3] function for the statement in Figure 3 is defined by the relation

$$\mathrm{S} = \{[\mathrm{time}, \mathrm{tri}] \rightarrow [0, \mathrm{time}, 1, \mathrm{tri}, 0]\},$$

where we have denoted that the first and fifth output tuple variables are equal to zero and the third is equal to one. A scheduling function maps iteration points of a specific loop nest to points in an iteration space for the whole computation [11, 1, 5, 3]. The lexicographic order of the points in the full iteration space defines the order of execution for the whole computation.

Figure 4 shows a transformed version of the original computation after applying three transformations: consecutive packing, locality grouping, and a partitioning. Consecutive packing (`cpack`) and locality grouping (`locgroup`) are data and iteration reordering heuristics used to improve data locality by optimizing the order of data accesses and were introduced by Ding and Kennedy [9]. In this example, we permute the data array `data[]` and permute the iterations of the tri loop based on the run-time computed permutations determined by `cpack` and `locgroup`. The partitioning introduces potential parallelism into the computation by adding a processor loop and a guard that ensures that the computation for each triangle is only executed on the processor that the triangle is mapped to (this is commonly referred to as an owner-computes strategy of parallelism). We note that the introduction of the processor 'guard' is sub-optimal. We are developing techniques for removing such guards, but the discussion of these techniques is beyond the scope of this paper. This loop could be annotated using an OpenMP `#pragma` to introduce parallelism across multiple processors. We represent these three transformations in the SPF using the following relations:

$$\begin{aligned}
\mathrm{R}_{\texttt{cpack}} &= \{[\mathrm{in}] \rightarrow [\mathrm{out}] : \mathrm{out} = \mathrm{sigma}(\mathrm{in})\}, \\
\mathrm{T}_{\texttt{locgroup}} &= \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{tri}, \mathrm{c}_2] \rightarrow [\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{k}, \mathrm{c}_2] : \mathrm{k} = \mathrm{delta}(\mathrm{tri})\}, \text{ and} \\
\mathrm{T}_{\texttt{part}} &= \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{tri}, \mathrm{c}_2] \rightarrow [\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{p}, \mathrm{c}_2, \mathrm{tri}, \mathrm{c}_3] : \mathrm{p} = \mathrm{proc}(\mathrm{tri})\}.
\end{aligned}$$

Note the use of the UFSs `sigma`, `delta`, and `proc` to represent mappings that will not be known until runtime.

2.3 The Apply Operation

Building on our definitions of sets and relations from the previous sections, next we describe three operations on sets and relations that are required by our transformation framework: first we discuss the apply operation in this section followed by the inverse and compose operations in Section 2.4. To generate code a transformation framework must be able to apply a schedule to a statement's iteration space to determine the statement's position in the context of the full computation. A statement's iteration space is represented with a set (e.g., $\mathrm{I} = \{[\mathrm{time}, \mathrm{tri}] : (0 \leq \mathrm{time} < \mathrm{T}) \wedge (0 \leq \mathrm{tri} < \mathrm{R})\}$). The scheduling function is represented as a relation (e.g., $\mathrm{S} = \{[\mathrm{time}, \mathrm{tri}] \rightarrow [0, \mathrm{time}, 1, \mathrm{tri}, 0]\}$). By applying the original scheduling function to the statement's iteration space, we can derive the full iteration space for that statement, including the nesting and ordering of that statement in relation to other statements. For example:

$$\begin{aligned}
\mathrm{F} &= \mathrm{S}(\mathrm{I}) \\
&= \{[\mathrm{time}, \mathrm{tri}] \rightarrow [0, \mathrm{time}, 1, \mathrm{tri}, 0]\}(\ \{[\mathrm{time}, \mathrm{tri}] : (0 \leq \mathrm{time} < \mathrm{T}) \wedge (0 \leq \mathrm{tri} < \mathrm{R})\}\ ) \\
&= \{[0, \mathrm{time}, 1, \mathrm{tri}, 0] : (0 \leq \mathrm{time} < \mathrm{T}) \wedge (0 \leq \mathrm{tri} < \mathrm{R})\}.
\end{aligned}$$

In general, the resulting set s of the apply operation $s = r_1(s_1)$ has the same tuple variables as the output tuple from the relation r_1, has constraints from both the relation r_1 and the input set s_1, and additionally has constraints that the input tuple variables from r_1 are pairwise equal to the tuple variables of s_1. Mathematically, the apply operation is defined as $(\vec{x} \in s) \iff (\exists \vec{y} \text{ s.t. } \vec{y} \in s_1 \wedge \vec{y} \rightarrow \vec{x} \in r_1)$. The general form of the apply operation is

$$\begin{aligned} s =& r_1(s_1) \\ =& \{[y_1, \ldots, y_m] \rightarrow [z_1, \ldots, z_n] : c_1 \wedge \ldots \wedge c_k\}(\\ & \quad \{[x_1, \ldots, x_m] : c_{k+1} \wedge \ldots \wedge c_{j+k+1}\} \ ) \\ =& \{[z_1, \ldots, z_n] : c_1 \wedge \ldots \wedge c_{k+j+1} \wedge x_1 = y_1 \wedge \ldots \wedge x_m = y_m\}. \end{aligned} \tag{5}$$

The apply operation is only legal when the arity of s_1 matches the input arity of r_1. The x_i and y_j variables become existentials in the resulting set, which later simplification will project out. If s_1 and/or r_1 involve more than one conjunction, then the above operation is performed on the Cartesian product of the two disjunctions of conjunctions. Note that in the following formulas, C_i and D_i denote conjunctions of constraints rather than single constraints:

$$r_1 = \{\vec{y} \rightarrow \vec{z} : C_1\} \vee \{\vec{y} \rightarrow \vec{z} : C_2\} \vee \ldots \vee \{\vec{y} \rightarrow \vec{z} : C_k\}, \tag{6}$$

$$s_1 = \{\vec{x} : D_1\} \vee \{\vec{x} : D_2\} \vee \ldots \vee \{\vec{x} : D_j\},$$

$$\begin{aligned} s =& r_1(s_1) \\ =& \{\vec{z} : C_1 \wedge D_1 \wedge \vec{x} = \vec{y}\} \vee \{\vec{z} : C_1 \wedge D_2 \wedge \vec{x} = \vec{y}\} \vee \ldots \vee \{\vec{z} : C_k \wedge D_j \wedge \vec{x} = \vec{y}\}. \end{aligned}$$

2.4 The Inverse and Compose Operations

We now describe both the inverse and compose operations. Just as with iteration spaces, a transformation framework must be able to place access functions within the whole context of the computation being represented and manipulated. The access function $\mathrm{A}_{\mathrm{orig}} = \{[\mathrm{time}, \mathrm{tri}] \rightarrow [\mathrm{k}] : \mathrm{k} = \mathrm{n1}(\mathrm{tri})\}$ is specified in terms of only the iterators of a single statement (in this case `time` and `tri`). Using the scheduling function for the statement (e.g., $\mathrm{S} = \{[\mathrm{time}, \mathrm{tri}] \rightarrow [0, \mathrm{time}, 1, \mathrm{tri}, 0]\}$) with the inverse and compose operations allows us to construct an access function that is in terms of the full iteration space `F`. The scheduling function is a relation from the iterations of a single statement to the full iteration space. We use the inverse operation to produce a new relation from the full iteration space to the iterators for the single statement. Finally, we compose the original access function (a relation from the single statement iterators to an array position) with this new relation to produce the full access function from the full iteration space to an array position:

$$\begin{aligned} \mathrm{A}_{\mathrm{full}} =& \mathrm{A}_{\mathrm{orig}}(\mathrm{S}^{-1}) \\ =& \{[\mathrm{time}, \mathrm{tri}] \rightarrow [\mathrm{k}] : \mathrm{k} = \mathrm{n1}(\mathrm{tri})\}(\ (\{[\mathrm{time}, \mathrm{tri}] \rightarrow [0, \mathrm{time}, 1, \mathrm{tri}, 0]\})^{-1} \ ) \\ =& \{[\mathrm{time}, \mathrm{tri}] \rightarrow [\mathrm{k}] : \mathrm{k} = \mathrm{n1}(\mathrm{tri})\}(\ \{[0, \mathrm{time}, 1, \mathrm{tri}, 0] \rightarrow [\mathrm{time}, \mathrm{tri}]\} \ ) \\ =& \{[0, \mathrm{time}, 1, \mathrm{tri}, 0] \rightarrow [\mathrm{k}] : \mathrm{k} = \mathrm{n1}(\mathrm{tri})\}. \end{aligned}$$

In general, the resulting relation r of the inverse operation $r = r_1^{-1}$ is a relation with the same constraints as r_1 but with input and output tuple variables swapped. Mathematically, the inverse operation is defined as $(\vec{x} \rightarrow \vec{y} \in r) \iff (\vec{y} \rightarrow \vec{x} \in r_1)$. In general, inverse has the form

$$\begin{aligned} r =& r_1^{-1} \\ =& (\{\vec{x} \rightarrow \vec{y} : C_1\} \vee \{\vec{x} \rightarrow \vec{y} : C_2\} \vee \ldots \vee \{\vec{x} \rightarrow \vec{y} : C_k\})^{-1} \\ =& \{\vec{y} \rightarrow \vec{x} : C_1\} \vee \{\vec{y} \rightarrow \vec{x} : C_2\} \vee \ldots \vee \{\vec{y} \rightarrow \vec{x} : C_k\}. \end{aligned} \tag{7}$$

The calculation of the full access function also utilizes the compose operation. The resulting relation r of the compose operation $r = r_1(r_2)$ has the input tuple variables from

r_2 and the output tuple variables from r_1, has constraints from both relations r_1 and r_2, and additionally has constraints that the output tuple variables from r_2 are pairwise equal to the input tuple variables from r_1: Mathematically, the compose operation is defined as $(\vec{x} \to \vec{y} \in r) \iff (\exists \vec{z} \text{ s.t. } \vec{x} \to \vec{z} \in r_2 \land \vec{z} \to \vec{y} \in r_1)$. In general, compose has the form

$$\begin{aligned} r =& r_1(r_2) \\ =&\{[d_1, \ldots, d_i] \to [e_1, \ldots, e_n] : c_{k+1} \land \ldots \land c_{j+k+1}\}(\\ &\quad \{[a_1, \ldots, a_m] \to [b_1, \ldots, b_i] : c_1 \land \ldots \land c_k\}) \\ =&\{[a_1, \ldots, a_m] \to [e_1, \ldots, e_n] : c_1 \land \ldots \land c_{k+j+1} \land b_1 = d_1 \land \ldots \land b_i = d_i\}. \end{aligned} \tag{8}$$

The compose operation is only legal when the output arity of r_2 is equal to the input arity of r_1. The b_i and d_i variables become existentials in the resulting set, which later simplification will attempt to project out. If r_1 and/or r_2 involve more than one conjunction, then the above operation is performed on the Cartesian product of the two collections of conjunctions as follows:

$$\begin{aligned} r_1 =& \{\vec{d} \to \vec{e} : F_1\} \lor \{\vec{d} \to \vec{e} : F_2\} \lor \ldots \lor \{\vec{d} \to \vec{e} : F_j\}, \\ r_2 =& \{\vec{a} \to \vec{b} : C_1\} \lor \{\vec{a} \to \vec{b} : C_2\} \lor \ldots \lor \{\vec{a} \to \vec{b} : C_k\}, \\ r =& r_1(r_2) \\ =& \{\vec{a} \to \vec{e} : C_1 \land F_1 \land \vec{b} = \vec{d}\} \lor \{\vec{a} \to \vec{e} : C_1 \land F_2 \land \vec{b} = \vec{e}\} \lor \\ & \ldots \lor \{\vec{a} \to \vec{e} : C_k \land F_j \land \vec{b} = \vec{d}\}. \end{aligned} \tag{9}$$

2.5 The Problems Arise: Transforming the Computation

This section presents two examples of situations where an operation produces a set or relation for which we are unable to generate code using existing code generation techniques. We have shown in Sections 2.1, 2.2, 2.3, and 2.4 how to represent the computation in Figure 3 and how to manipulate this representation to prepare it for transformation. Now we show that applying the transformations $\mathtt{R_{cpack}}$, $\mathtt{T_{locgroup}}$, and $\mathtt{T_{part}}$ to the data and the computation results in sets and relations that inhibit code generation.

We first apply the consecutive packing transformation to the computation. This involves composing the access function transformation (e.g., $\mathrm{R}_{\mathtt{cpack}} = \{[\mathrm{in}] \to [\mathrm{out}] : \mathrm{out} = \mathrm{sigma}(\mathrm{in})\}$) with the full access function for the statement (e.g., $\mathrm{A}_{\mathrm{full}} = \{[0, \mathrm{time}, 1, \mathrm{tri}, 0] \to [\mathrm{k}] : \mathrm{k} = \mathrm{n1}(\mathrm{tri})\}$) to derive the transformed access function:

$$\begin{aligned} \mathrm{A}'_{\mathrm{full}} =& \mathrm{R}_{\mathtt{cpack}}(\mathrm{A}_{\mathrm{full}}) \\ =& \{[0, \mathrm{time}, 1, \mathrm{tri}, 0] \to [\mathrm{out}] : \mathrm{out} = \mathrm{sigma}(\mathrm{n1}(\mathrm{tri}))\}. \end{aligned}$$

When applying an iteration permutation RTRT such as locality grouping, we must update the access functions of the statements that are within the loop whose iterations are being permuted using a transformation relation (e.g., $\mathrm{T}_{\mathtt{locgroup}} = \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{tri}, \mathrm{c}_2] \to [\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{p}, \mathrm{c}_2, \mathrm{tri}, \mathrm{c}_3] : \mathrm{p} = \mathrm{proc}(\mathrm{tri})\}$). As an example, consider the following operations that update $\mathrm{A}'_{\mathrm{full}}$:

$$\begin{aligned} \mathrm{A}''_{\mathrm{full}} =& \mathrm{A}'_{\mathrm{full}}(\mathrm{T}^{-1}_{\mathtt{locgroup}}) \\ =& \{[0, \mathrm{time}, 1, \mathrm{tri}, 0] \to [\mathrm{out}] : \mathrm{out} = \mathrm{sigma}(\mathrm{n1}(\mathrm{tri}))\}(\\ &\quad \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{k}, \mathrm{c}_2] \to [\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{tri}, \mathrm{c}_2] : \mathrm{k} = \mathrm{delta}(\mathrm{tri})\}) \\ =& \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{k}, \mathrm{c}_2] \to [\mathrm{out}] : \mathrm{k} = \mathrm{delta}(\mathrm{tri}) \land \mathrm{out} = \mathrm{sigma}(\mathrm{n1}(\mathrm{tri}))\}. \end{aligned}$$

Notice this operation results in the presence of the existential `tri` as the input to the UFS `delta`. This is a problem as existing techniques such as Fourier-Motzkin do not support projecting out existentials that are inputs to UFSs. To generate efficient code for this access relation, we need the output tuple variable `out` expressed as a function of the input tuple variables.

Finally, we apply a partitioning transformation (e.g., $\mathrm{T}_{\texttt{part}} = \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{tri}, \mathrm{c}_2] \to [\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{p}, \mathrm{c}_2, \mathrm{tri}, \mathrm{c}_3] : \mathrm{p} = \mathrm{proc}(\mathrm{tri})\}$) to the computation by updating the iteration spaces of the affected statements (e.g., $\mathrm{F} = \{[0, \mathrm{time}, 1, \mathrm{tri}, 0] : (0 \le \mathrm{time} < \mathrm{T}) \wedge (0 \le \mathrm{tri} < \mathrm{R})\}$):

$$\begin{aligned}\mathrm{F}' =&\mathrm{T}_{\texttt{part}}(\mathrm{F})\\ =&\{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{tri}, \mathrm{c}_2] \to [\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{p}, \mathrm{c}_2, \mathrm{tri}, \mathrm{c}_3] : \mathrm{p} = \mathrm{proc}(\mathrm{tri})\}(\\ &\quad\{[0, \mathrm{time}, 1, \mathrm{tri}, 0] : (0 \le \mathrm{time} < \mathrm{T}) \wedge (0 \le \mathrm{tri} < \mathrm{R})\})\\ =&\{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{p}, \mathrm{c}_2, \mathrm{tri}, \mathrm{c}_3] : (0 \le \mathrm{time} < \mathrm{T}) \wedge (0 \le \mathrm{tri} < \mathrm{R}) \wedge \mathrm{p} = \mathrm{proc}(\mathrm{tri})\}.\end{aligned}$$

Note the fact that the iterator `p` does not have explicit affine bounds but rather is equal to the output of the UFS `proc`. In order to generate code that iterates over all points in this set, we need affine bounds for each iterator/tuple variable.

The application of both the locality grouping and partitioning transformations resulted in situations where code generation is inhibited. In the first case, the resulting access function contains a constraint where an existential is an input to a UFS. In the second case, we do not have explicit bounds on a loop iterator. These types of situations inhibit the generation of polyhedral scanning loop nests by tools such as CLooG [3, 4, 5] or the generation of expressions for array accesses based on access functions. Section 3 introduces techniques to rectify these situations to enable code generation.

3 Constraint Simplification in the SPF

Section 2 showed examples of how a computation can be represented and transformed using the Sparse Polyhedral Framework. After applying RTRTs to the computation it is possible to produce constraints that contain existentials as inputs to UFSs (e.g., $\texttt{A}''_{\texttt{full}}$) or tuple variables without explicit affine bounds (e.g., $\texttt{F}'$)—situations that inhibit code generation. This section introduces remedies for these situations.

We identify three general situations where code generation is inhibited. In the following, let v be an existential, let t be a tuple variable, let f be a UFS:

- $t = f(v)$: An existential is an input to a UFS,
- $v = \mathrm{expr}$: An existential is equal to an affine expression, possibly containing a UFS instance, and
- $t = f(\ldots)$: A tuple variable is equal to a UFS instance and no affine bounds are present for that variable in other constraints.

These situations were generalized from specific instances of each that arose during our work with the SPF on an example benchmark. This list is thus necessary but may not be sufficient for enabling code generation—there may be other situations involving UFSs that we have not encountered that make it difficult to generate code. We examine each of these general cases in the following subsections.

3.1 Inverse Function Simplification

When performing operations on sets and relations to represent and transform computations in the SPF, it is possible to introduce an existential that is an input to a UFS. We have shown that this is possible by applying the locality grouping iteration permutation RTRT to our example computation. Specifically, we produced the following access function:

$$\mathrm{A}''_{\mathrm{full}} = \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{k}, \mathrm{c}_2] \to [\mathrm{out}] : \mathrm{k} = \mathrm{delta}(\mathrm{tri}) \wedge \mathrm{out} = \mathrm{sigma}(\mathrm{n1}(\mathrm{tri}))\}.$$

Notice in the constraint $\mathrm{k} = \mathrm{delta}(\mathrm{tri})$ that the argument to the function `delta`, `tri`, is an existential. Therefore we need to manipulate these constraints in some way so that we can remove the existential.

If we are given the fact that the UFS `delta` is invertible (with its inverse named $\texttt{delta}^{-1}$ for example), we can utilize this information to remove the existential from the constraints. More specifically, we can utilize information that `delta` is a bijection to perform an obvious simplification on the equality constraint. We often deal with permutations such as those that are the result of `cpack` or `locgroup`. Since permutations are bijections, they can be inverted automatically at runtime. For this simplification to apply, we only require to know that a given UFS is a bijection and what the inverse of the UFS is named.

When we apply this simplification rule to the constraint in the relation $\mathrm{A}''_{\mathrm{full}}$, we obtain the following relation:

$$\mathrm{A}''_{\mathrm{full}} = \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{k}, \mathrm{c}_2] \rightarrow [\mathrm{out}] : \mathrm{delta}^{-1}(\mathrm{k}) = \mathrm{tri} \wedge \mathrm{out} = \mathrm{sigma}(\mathrm{n1}(\mathrm{tri}))\}.$$

This relation is now one step closer to having all existentials removed from the constraints. At this point, assuming we knew `sigma` and `n1` were also bijections, we could apply the inverse simplification rule twice to the constraint `out` = `sigma`(`n1`(`tri`)) to obtain the constraint $\texttt{n1}^{-1}(\texttt{sigma}^{-1}(\texttt{out})) = \texttt{tri}$. In order to generate code, however, we require constraints be in terms of the output tuple variable `out`. Additionally, the permutability of `n1` is not known and therefore we could not apply this rule for this UFS to expose the existential `tri`. The next section will discuss how we completely remove the existential `tri` from the remaining constraints using a different approach.

In general, the inverse function simplification applies to equality constraints of the form:

$$t = f(v),$$

where t is a tuple variable, v is an existential, and we know that f is a function that admits an inverse and that inverse is f^{-1}. In this situation, we can transform the constraint into:

$$f^{-1}(t) = v,$$

to produce a constraint with tuple variable t as input to the inverse function f^{-1}.

We show that this is a legal constraint manipulation using the fact that f admits an inverse:

$$\begin{aligned} t =& f(v) \\ \implies \quad f^{-1}(t) =& f^{-1}(f(v)) \\ \implies \quad f^{-1}(t) =& v. \end{aligned} \tag{10}$$

3.2 Existential Equality Simplification

We utilize a second simplification rule to remove existentials involved in equalities with UFSs. The use of the inverse function simplification in the previous section produced the following relation:

$$\mathrm{A}''_{\mathrm{full}} = \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{k}, \mathrm{c}_2] \rightarrow [\mathrm{out}] : \mathrm{delta}^{-1}(\mathrm{k}) = \mathrm{tri} \wedge \mathrm{out} = \mathrm{sigma}(\mathrm{n1}(\mathrm{tri}))\}.$$

This relation still contains the existential `tri` We remove this variable using the existential equality simplification. If an existential is present in a equality constraint, we can remove that constraint and replace all instances of that variable's equivalent value. In this example, we replace tri with $\mathrm{delta}^{-1}(\mathrm{k})$ to produce the following relation:

$$\mathrm{A}''_{\mathrm{full}} = \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{k}, \mathrm{c}_2] \rightarrow [\mathrm{out}] : \mathrm{out} = \mathrm{sigma}(\mathrm{n1}(\mathrm{delta}^{-1}(\mathrm{k})))\}.$$

After applying this simplification, we have removed all existentials from the constraints of this relation and thus may generate code for this access function.

In general, the existential equality simplification can be used with equality constraints of the form

$$v = \mathrm{expr},$$

where v is an existential and expr is any expression, possibly involving UFSs (though this is not required). If an equality constraint of this form is found, we can:

1. remove the equality constraint from the collection of constraints of the set or relation and

2. replace all instances of the existential v with the expression that v is equal to (expr).

The legality of this simplification can be ensured by considering the fact that we apply this simplification to equality constraints. Since the existential in question is equal to the expression we are replacing it with, we know that simply removing that constraint and modifying all other constraints that reference this existential is legal. Note that we do not apply this simplification rule in situations where the existential in question is not used in other constraints.

3.3 Affine Approximation

In this section, we introduce a method for introducing affine bounds for tuple variables that are constrained by an equality with a UFS. Consider the following full iteration space that we derived at the end of Section 2.5:

$$\mathrm{F}' = \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{p}, \mathrm{c}_2, \mathrm{tri}, \mathrm{c}_3] : (0 \leq \mathrm{time} < \mathrm{T}) \wedge (0 \leq \mathrm{tri} < \mathrm{R}) \wedge \mathrm{p} = \mathrm{proc}(\mathrm{tri})\}.$$

One method of code generation requires that all iterators of an iteration space have affine bounds. However, note that the tuple varible `p` is only involved in an equality constraint that contains a UFS.

To fix the situation when a tuple variable has no explicit affine bounds, we utilize additional information about the UFS to which it is equal. Specifically, we note that if we know the range of the UFS `proc` in the above example, we can introduce inequalities that bound the tuple variable `p`:

$$\begin{aligned}\mathrm{F}' =& \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{p}, \mathrm{c}_2, \mathrm{tri}, \mathrm{c}_3] : \\ & (0 \leq \mathrm{time} < \mathrm{T}) \wedge (0 \leq \mathrm{tri} < \mathrm{R}) \wedge \mathrm{p} = \mathrm{proc}(\mathrm{tri}) \wedge (0 \leq \mathrm{p} < \mathrm{n\_procs})\}.\end{aligned}$$

Here we have introduced two inequalities that bound `p` from above and below. We utilize the information that the UFS `proc` ranges from `0` to `n_procs` to allow us to approximate the value of `p`. After this approximation, we are able to generate a loop that scans the range of possible values for `p`.

In general, suppose we have a constraint of the form

$$t = f(...),$$

where t is a tuple variable, f is some UFS for which we have compile-time range information and no other affine bounds are available for t. In this case, since we know the range of f, we can add approximate constraints on t. Specifically, we can conservatively say that t will fall somewhere within the upper and lower bounds of the range of f, or

$$\mathrm{lower\_bound}(\mathrm{range}(f)) \leq t \leq \mathrm{upper\_bound}(\mathrm{range}(f))$$

in constraint form. We may safely introduce these inequality constraints without changing meaning of the set or relation since we are only bounding the values of the tuple variable t and not restricting its value beyond the original equality constraint.

3.4 Simplification Algorithm

In the previous sections we described three techniques for manipulating set and relation constraints that are required to enable code generation of the transformed code in Figure 4. Our approach for applying these three techniques as a complete simplification algorithm is shown as Algorithm 1. Our implementation uses this algorithm in two situations: at the time a set or relation is constructed from a user's specification and on a set or relation that is the result of an operation such as apply, inverse, or compose.

This algorithm attempts to successively project out each existential present in a set or relation's constraints. For each existential we attempt to apply the inverse function simplification and existential equality simplification as many times as they are applicable. Once these have been applied, we check that no existential is present in constraints containing a UFS. We then use Fourier-Motzkin elimination to project the existential out of any remaining affine constraints. Finally, we apply the affine approximation technique to introduce affine bounds for tuple variables where possible.

```
for each existential v do
    Inverse function simplifications;
    Existential equality simplifications;
    if v present in a UFS constraint then
        fail
    end
    Project out v from the constraints using Fourier-Motzkin;
end
Affine approximation;
```

Algorithm 1: Set/Relation Simplification Algorithm

4 Evaluation

In this section, we present a discussion concerning the complexity and performance of the set and relation operations and simplifications. We first discuss the complexity of each operation and the simplification algorithm presented in Section 3.4. The second half of this section will present the performance of our implementation of the set and relation concepts from this paper.

4.1 Operation and Simplification Complexity Analysis

Understanding the algorithmic complexity of the set and relation operations and simplifications allows us to have some idea for the upper bound on the expected running time of a code generation tool based on our techniques. We first present the complexity of the constraint simplification techniques followed by the overall complexity of the set/relation operations. For the following discussion, take N to be the number of constraints and M to be the number of conjunctions.

In general when performing an operation such as union or compose, we first perform the steps required by the operation to produce a new set of constraints. Following this we apply our simplification techniques to attempt to project out existentials from the constraints. Therefore, the complexity of each operation is a function both of the operation itself and of simplification. Since simplification is common to all operation complexity, we discuss this first.

We have discussed our simplification algorithm in Section 3.4 and have presented it as Algorithm 1. For each existential present in a set or relation's constraints, this algorithm attempts to project out that variable. It first applies the inverse function simplification and the existential equality simplification to remove existentials from constraints containing UFSs. It then uses Fourier-Motzkin elimination to project the variable out of the constraints that do not contain UFSs. The complexity of Fourier-Motzkin elimination is double exponential in the number of constraints and variables. Therefore, this is the dominating term for our simplification algorithm. However, for the sake of completeness, we note that both the inverse function simplification and the existential equality simplification are $O(N)$ for each existential since we must perform a linear scan through all constraints. Also, as an optimization, we note that we can utilize the existential equality simplification when projecting out existentials involved in equality constraints. We have found that this optimization avoids running full FM in the majority of cases, avoiding the worst-case exponential complexity of FM.

The complexity of both the apply and compose operations is $O(M^2)$ in the total number of conjunctions since we must create a new conjunction for every pair of conjunctions between operands. Union is $O(M)$ since we must combine all conjunctions into a single set or relation. Inverse is $O(1)$ since we only change the order of the tuple variables and no manipulation of the conjunctions or constraints is necessary.

4.2 Implementation Performance

We evaluate the performance and power of our set and relation implementation by comparing to Omega [11, 10] where possible. We base our evaluation on a suite of set/relation operations taken from a real-world run of our tool, IEGen. This run consisted of the application of three transformations (`cpack`, `locgroup`, and a loop alignment) and one optimization (pointer update, or collapsing nested UFSs) to a molecular dynamics benchmark. The benchmark itself, `moldyn` [14], consisted of approximately 100 lines of C code with three main loops nested within an outer timestep loop. We extracted only the sets and relation operations from this run to form our suite of operations for benchmarking. We measured various aspects of performing these operations after constructing the arguments to each operation. We first present information about these operations only related to our implementation followed by a discussion of our implementation compared to Omega.

The whole run of IEGen of the transformation sequence on the `moldyn` computation took a total of 1.85 seconds. Running just the set and relation operations that this consisted of (the operations suite) took 1.46 seconds. Figure 5 presents the number of each operation type for 77 total operations that the operations suite contains. Figure 6 shows a histogram of the numbers of tuple variables in the results of the operations. Figure 7 shows a histogram of the numbers of existentials that need to be projected out for the operations. The time to run each of the collections of individual operation types is shown in Figure 8. Across all operations, a total of 246 existentials were projected out. Most of these were projected out using one or more applications of the existential equality simplification. Only 16 existentials required using the full Fourier-Motzkin algorithm to be projected out of the constraints. The inverse simplification rule was applied a total of 30 times.

Omega's support for UFSs is limited [18, 23]. Syntatically, it requires that inputs to UFSs be prefixes of the tuple variables. Additionally, any attempt to project out an existential that is the input to a UFS results in an UNKNOWN constraint being added to the result signifying that the result is only an approximation and may not be correct. Due to these restrictions, Omega is not able to perform all of the 77 operations that the operations suite contains. We filtered out the operations that Omega could not support to create a subset of operations that Omega can support. Of the 136 sets and relations that were arguments to the operations 81 did not contain UFSs. The total number of operations where all arguments contain no UFSs is 33, composed of:

- Inverse: 4
- Apply: 16
- Compose: 4
- Set Union: 9

We use these 33 operations to compare our implemementation performance to Omega's. Figure 9 presents the times taken for our implementation and Omega to perform all operations from each of the four types above. In most cases we are two orders of magnitude slower than Omega. We note that our initial implementation goal was not raw performance but rather supporting the operations and simplifications required for code generation. To improve the performance, we could utilize techniques implemented in Omega [17]. The performance goal for our implementation is to be fast enough to support interactive use of IEGen. For moldyn, we've come very close to meeting this goal, but there is clearly still room for improvement.

5 Related Work

Our work on the Sparse Polyhedral Model builds upon many ideas and projects within the field of program optimization. This section describes how the work we present in this paper relates to two areas of work: libraries for program representation and tools for program optimization and code generation. We use the techniques in this paper to implement a library for representing programs within the SPF. Additionally, we are developing a tool that uses this library to generate optimized irregular computations.

We note four polyhedral libraries that provide representations and operations on polyhedra: Omega, Polylib, PPL, and ISL. The Omega project [11, 10] is most closely related to the work we present in this paper. Used primarily for data dependence analysis, Omega's usage of sets and relations (based on Presburger arithmetic) is the same mathematical concept that our work is based on. The Omega library provides an implementation of sets and relations with limited support for UFSs. Unlike the techniques we have presented in this paper, Omega does not utilize any additional information about the UFSs. Therefore, the library is unable to perform operations like the inverse function simplification or affine approximation.

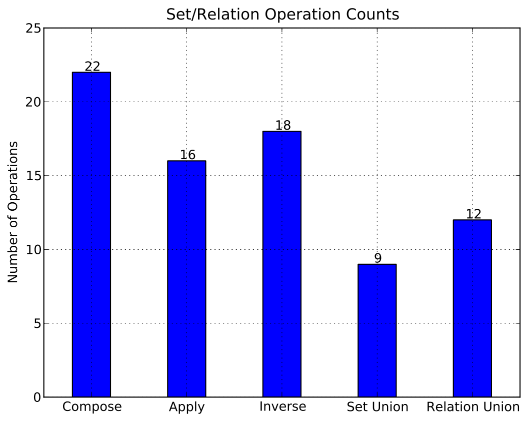

Figure 5: Counts of number of operations of 5 types in the operations suite

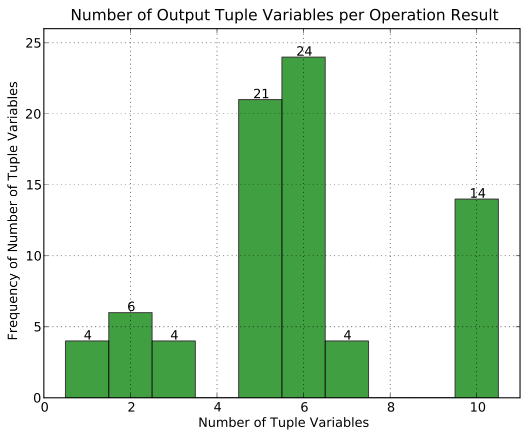

Figure 6: Histogram of numbers of tuple variables in the results of all operations in the operations suite

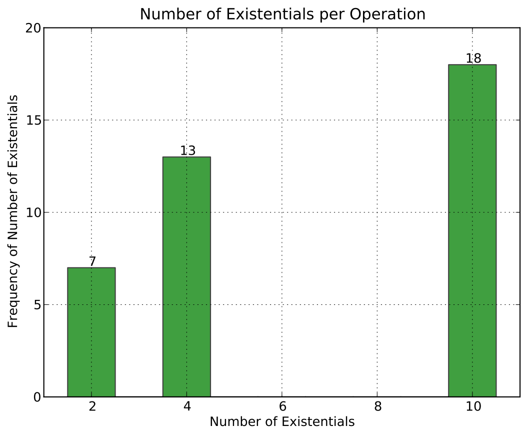

Figure 7: Histogram of total number of existentials for all operations in the operations suite

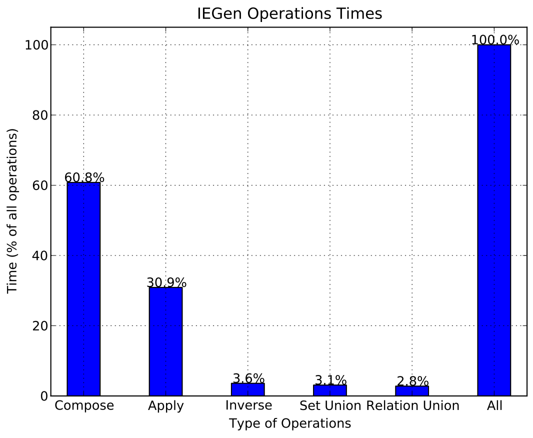

Figure 8: Times for performing various collections of operations in the operations suite grouped by type

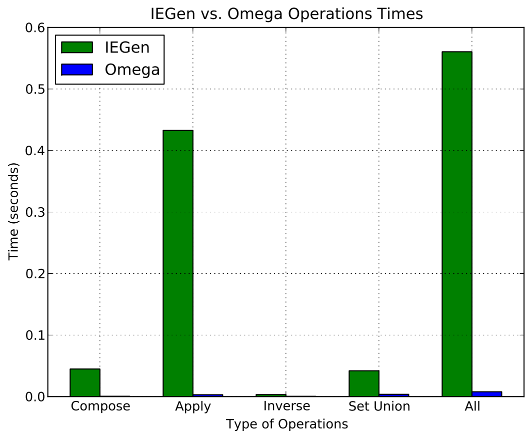

Figure 9: Times for performing various collections of operations in that Omega could run

Other limitations are that inputs to uninterpreted functions in Omega can only be prefixes of input and/or output tuples and composition of two relations with uninterpreted functions results in an UNKNOWN constraint. To a limited extent, uninterpreted function symbols have been used in Omega to represent indirect array accesses for determining more accurate data dependence information.

Polylib [22, 12] is a library that provides a way to represent and manipulate unions of parameterized polyhedra. Polylib maintains both constraint and generator representations of the polyhedra as some operations are more efficient on one representation than the other. Our implementation uses an approach similar to the constraint representation. Polylib does not have support for UFSs and thus is not able to support the class of programs we are targeting. Another library that represents polyhedral is PPL [2]. Similarly to Polylib, PPL maintains dual representations of polyhedra. Just as with Polylib, PPL represents rational polyhedra as well. PPL has no support for representing UFSs. Finally, ISL [21] is a fourth library that represents polyhedra. This library recently became the default polyhedral backend for CLooG.

CLooG, Pluto, Graphite, FADA, and PoCC are tools used for code generation, program analysis, and optimization that utilize one or more of the polyhedral libraries. CLooG [3, 4, 5] is a tool for generating code that scans a specified union of polyhedra. We use CLooG for generating loop nests that scan the iteration spaces of our computations. Pluto [7] is an automatic polyhedral loop optimizer and parallelizer that utilizes a large set of tools from the polyhedral community including CLooG for generating scanning loop nests. Graphite [15] is used as a polyhedral optimizing backend for the popular GCC compiler suite. FADA [8, 6] (Fuzzy Array Dataflow Analysis) is a tool for performing dataflow analysis on irregular programs. It performs an 'instance wise dependence analysis for non-static control programs'. The FADA Toolkit is a partial implementation of this analysis. FADA offers full support for static control programs as it extends Feautrier's Array Dataflow Analysis (ADA). Additionally, FADA supports while loops and conditionals, scalar and array references,and indirect array accesses (such as we require in SPF) and non-affine array accesses. FADA does not support pointer indirection and array cell aliasing. The output from FADA is a set of dataflow dependences in the form of quasts (quasi-affine selection tree). Our work may be able to utilize a tool like the FADA Toolkit to support dataflow analysis to determine if a specified transformation is legal. Finally, PoCC [16] (the Polyhedral Compiler Collection) is a source-to-source optimizing compiler.

6 Conclusions

Many scientific computations utilize indirect array accesses and therefore require inspector/executor strategies for performing run-time reordering transformations. The Sparse Polyhedral Framework (SPF) is an extension to the polyhedral model that supports the compile-time application of run-time reordering transformations (RTRTs) and the generation of inspectors and executors that implement the RTRTs. The key addition of the SPF is the ability to utilize sets and relations with uninterpreted function symbols (UFSs) in constraints at compile time to represent run-time entities such as index arrays and run-time reorderings. To generate code for iteration sets and array access functions that involve UFS constraints, it is necessary to project existentials out of such constraints and provide all of the loop iterators with affine constraints. We present three simplification rules that if applied to these situations, produce sets and relations for which we are able to generate code. We evaluate our implementation of these techniques by comparing the functionality and performance to Omega. Using the sets and relations that support UFSs enables us to develop a transformation framework and programming model that supports the application of RTRTs to irregular codes.

References

[1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. *Int. J. Parallel Program.*, 29(5):493–544, 2001.

[2] R. Bagnara, P. M. Hill, and E. Zaffanella. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008.

[3] C. Bastoul. Generating loops for scanning polyhedra. Technical Report 2002/23, PRiSM, Versailles University, 2002.

[4] C. Bastoul. Efficient code generation for automatic parallelization and optimization. pages 23–30, october 2003.

[5] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th Interntional Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2004.

[6] M. Beloucha, D. Barthou, and S.-A.-A. Touati. *FADA Toolkit Users Guide*. University of Versailles Saint-Quentin en Yvelines, France, October 2009.

[7] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6):101–113, 2008.

[8] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 92–101, New York, NY, USA, 1995. ACM.

[9] 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 Conference on Programming Language Design and Implementation*, pages 229–241, Atlanta, Georgia, May 1999.

[10] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *The Omega Calculator and Library*, November 1996.

[11] W. Kelly and W. Pugh. A unifying framework for iteration reordering transformations. Technical report, University of Maryland, College Park, MD, USA, October 1995.

[12] V. Loechner. Polylib: A library for manipulating parameterized polyhedra, 1999.

[13] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. Principles of runtime support for parallel processors. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 140–152, New York, NY, USA, 1988. ACM.

[14] R. Ponnusamy, Y.-S. Hwang, R. Das, J. Saltz, A. Choudhary, and G. Fox. Supporting irregular distributions in fortran 90d/hpf compilers. Technical report, College Park, MD, USA, 1994.

[15] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G. A. Silber, and N. Vasilache. Graphite: Loop optimizations based on the polyhedral model for gcc. In *Proc. of the 4th GCC Developper's Summit*, pages 179–198, June 2006.

[16] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Hybrid iterative and model-driven optimization in the polyhedral model. Technical Report 6962, INRIA Research Report, June 2009.

[17] W. Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, 1992.

[18] W. Pugh and D. Wonnacott. Nonlinear array dependence analysis. Technical Report CS-TR-3372, College Park, MD, USA, November 1994.

[19] M. M. Strout. Compile-time composition of run-time data and iteration reorderings. In *In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[20] M. M. Strout, L. Carter, and J. Ferrante. Rescheduling for locality in sparse matrix computations. In *Proceedings of the 2001 International Conference on Computational Science, Lecture Notes in Computer Science*, pages 28–30. Springer-Verlag, 2001.

[21] S. Verdoolaege. An integer set library for program analysis. In *Advances in the Theory of Integer Linear Optimization and its Extensions, AMS 2009 Spring Western Section Meeting*, San Francisco, California, April 2009.

[22] D. K. Wilde. A library for doing polyhedral operations. Technical report, 1993.

[23] D. G. Wonnacott. *Constraint-Based Array Dependence Analysis.* PhD thesis, University of Maryland, College Park, 1995.