Computer Science

Technical Report

Colorado State University

Value-based Dependence Analysis for the $\mathcal{Z}$-ployhedral Model

DaeGon Kim and Gautam and Sanjay V. Rajopadhye

[kim|ggupta|svr]@cs.colostate.edu

April 17, 2008

Technical Report CS-08-100

Computer Science Department

Colorado State University

Fort Collins, CO 80523-1873

Phone: (970) 491-5792 Fax: (970) 491-2466

WWW: http://www.cs.colostate.edu

Value-based Dependence Analysis for the $\mathcal{Z}$-ployhedral Model

DaeGon Kim and Gautam and Sanjay V. Rajopadhye

[kim|ggupta|svr]@cs.colostate.edu

April 17, 2008

Abstract

Data dependence analysis is a crucial step for any computation reordering transformation. Exposing the inherent ordering among the computations in different loop iterations, it provides the conditions for the validity of loop transformations such as automatic parallelization and tiling.

Exact dependence analysis gives the minimal set of computation orderings to be respected by transformations. Currently, a restricted class of programs, called affine control loops (ACLs), can be analyzed to produce exact dependence relations. In this paper we extend this analysis to a more general class of programs that contain (*i*) loops with non-unit stride, (*ii*) modulo operations, integer divisions, max and min in: loop bounds, guards, and memory access functions, and (*iii*) existential variables in guards. More precisely, the proposed technique can be applied to loop programs having statements defined over Presburger sets. The results of our analysis are programs in the $\mathcal{Z}$-polyhedral model, a generalization of the well-established polyhedral model. The beauty of our formulation is that with straightforward pre- and post-processing we can reuse the existing tools and techniques previously developed for ACLs.

1 Introduction

A significant fraction of program execution time is spent in loops, thus, most optimizing compilers focus on their efficient compilation. Various loop transformations such as reordering, automatic parallelization or blocking have been developed for almost three decades. To ensure the semantic equivalence of any transformation, it is essential to respect the dependences between statement instances. The extraction of these dependences is what is known as *value-based dependence analysis* or simply *dependence analysis*. Since dependences between statement instances dictate the space of valid transformations, the power of loop transformations relies critically on the chosen iteration space and dependence abstraction. The most sophisticated existing technology is the polyhedral model that enables the analysis and transformation of loops that have a polyhedral iteration space and affine dependences between different statement instances. In this paper we present the theory for the exact dependence analysis of a more general class of loops, having statements defined over richer domains. As such, we enable much stronger optimizations.

Example 1 *highlights the need for our result. Given the following sequential loop*

```
for i = 1 ... n
  if ((i%2==0)||(i%3==0))
    X[i] = f(X[i-1]);
```

note that the statement is not executed at all loop iterations. When $i = 6j+1$ *or* $i = 6j+5$ *for any integer* j*, the statement guard* `((i%2==0)||(i%3==0))` *fails. However, any polyhedral approach is incapable of factoring this information, and therefore, conservatively assumes that the statement* `X[i] = f(X[i-1])` *is executed at all loop iterations. As all statements instances depend on the value obtained from the previous iteration, this abstraction clearly imposes a sequential execution order of the loop. Moreover, if we were to use the more precise information, we may obtain the constant-time parallelization of this* $\Theta(n)$ *loop given below.*

```
forall i = 2 ... n step 6
  X[i] = f(X[i-1]);
forall i = 6 ... n step 6
  X[i] = f(X[i-1]);
forall i = 3 ... n step 6
  X[i] = f(X[i-1]);
forall i = 4 ... n step 6
  X[i] = f(X[i-1]);
```

The iteration spaces of the each of the four loops are similar to the original loop, but restricted to certain periodic hops. More precisely, it is the intersection of the original polyhedron $i \in \{1, \ldots, n\}$ and a lattice (with a periodicity of 6). Such a set is called a $\mathcal{Z}$-polyhedron. Broadly, the shown parallelization requires the dependence analysis, scheduling analysis [13] and code generation [1] of loops having statements defined over $\mathcal{Z}$-polyhedral iteration domains.

The loops that have a precise abstraction in the polyhedral model are called affine control loops or ACLs. As mentioned, exact dependence analysis has been developed for ACLs [8, 25, 24]. Based on the abstraction, sophisticated techniques such as memory optimization [6, 17, 26, 20, 31, 4], automatic parallelization [2, 3, 9, 10, 15, 16, 21, 23, 27, 28, 30] and, more recently, the automatic and optimal decrease in the asymptotic complexity of accumulations [12] have been developed in the polyhedral model.

However, there are many important loop programs that are not ACLs. Loops with non-unit stride are required to express computations such as the RED-BLACK SOR for solving partial differential equations. More importantly, tiled programs generally encode information of the tile origins and lattice. Any subsequent analysis of such programs, primarily for the purpose of porting to different architectures requires the more general dependence analysis presented here. For similar mathematical reasons, non-unimodular transformations [11, 19, 32, 29], used for program optimization and in the derivation of parallel architectures with

periodic processor activity such as *multi-rate arrays* [18] and bidirectional systolic arrays result in loops that are not ACLs, thus, cannot be expressed in the polyhedral model.

We present the dependence analysis for loops having statements defined over $\mathcal{Z}$-polyhedral domains. In fact, we will accept loops having statements that are defined over arbitrary Presburger sets and use our previous result that shows the equivalence of Presburger sets and unions of $\mathcal{Z}$-polyhedra. It is precisely this result that has enabled the extension proposed here. The decidable theory of Presburger formulae has been known for many decades which has also been incorporated in the Omega tool [25] used extensively in optimizing compilers. In spite of the greater generality of the underlying machinery, the Omega test is often restricted to ACLs in practice.

In this paper we present the exact dependence analysis technique for a richer class of programs where statements can be defined over Presburger sets. Our main contributions are

- The extension of the exact data-flow analysis to a more general class of programs that include non-unit stride loops and have mod, integer division, max and min operations in loop bounds, guards and even memory access functions. These statements can be further restricted by guards that have existentially quantified variables.

- A novel ILP formulation that enables us to reuse the existing tools and techniques with a simple pre- and post-processing.

The remainder of this paper is organized as follows. In the following section, we provide the required mathematical background for our technique. In Section 3, we illustrate our approach with an example. In Section 4 we describe our input language, precisely characterize the iteration space of statements, and describe the data-dependence problem of obtaining the producer for values read by any statement. We then express this problem as an ILP formulation. Our iteration spaces are Presburger sets which may be expressed as a union of $\mathcal{Z}$-polyhedra. Finally, we post-process the results of the ILP solver to obtain an answer in terms of iterations of the $\mathcal{Z}$-polyhedron. After discussing related work in Section 5, we

finally present our conclusions.

2 Mathematical Background

In this section, we will define the mathematical objects and concepts used in our analysis. As a convention, we will denote matrices with the upper-case letters and vectors with the lower-case. All matrices and vectors have integer elements. We will denote the identity matrix by I. Syntactically, the different elements of a vector v will be written as a list.

2.1 Affine Lattices

The lattice generated by a matrix L is the set of all integer linear combinations of the columns of L. If the columns of a matrix are linearly independent, they constitute a *basis* of the generated lattice.

We use a generalization of the lattices generated by a matrix, additionally allowing offsets by constant vectors. These are called *affine lattices*. An affine lattice is a subset of $\mathbb{Z}^n$ and can be represented as $\{Lz + l | z \in \mathbb{Z}^m\}$ where L and l are an $n \times m$ matrix and n-vector respectively. We call z the coordinates of the affine lattice.

2.2 Integer Polyhedra

An *integer polyhedron*, $\mathcal{P}$ is a subset of $\mathbb{Z}^n$ that can be defined by a finite number of affine inequalities (also called affine constraints or just constraints when there is no ambiguity) with integer coefficients. We follow the convention that the affine constraint c_i is given as $(a_i^T z + \alpha_i \geq 0)$ where $z, a_i \in \mathbb{Z}^n, \alpha_i \in \mathbb{Z}$. The integer polyhedron, $\mathcal{P}$, satisfying the set of constraints $\mathcal{C} = \{c_1, \ldots, c_b\}$ is often written as $\{z \in \mathbb{Z}^n | Qz + q \geq 0\}$ where $Q = (a_1 \ldots a_b)^T$ is an $b \times n$ matrix and $q = (\alpha_1 \ldots \alpha_b)^T$ is an b-vector eg. $\{i, j | 0 \leq i, 0 \leq j\}$ is the polyhedron corresponding to the first orthant. A parameterized integer polyhedron is an integer polyhedron where some indices are interpreted as size parameters.

2.3 $\mathcal{Z}$-Polyhedra

A $\mathcal{Z}$-polyhedron is the intersection of an integer polyhedron and an affine lattice. We use the following representation of $\mathcal{Z}$-polyhedra.

$$\{Lz + l | Qz + q \geq 0, z \in \mathbb{Z}^m\} \tag{1}$$

where the columns of L constitute a basis of the affine lattice $\{Lz+l|z \in \mathbb{Z}^m\}$ and valid values of z are given by the *coordinate polyhedron* $\{z|Qz + q \geq 0, z \in \mathbb{Z}^m\}$ of the $\mathcal{Z}$-polyhedron. Iteration points of the $\mathcal{Z}$-polyhedral domain are points of the affine lattice corresponding to valid coordinates.

A parameterized $\mathcal{Z}$-polyhedron is a $\mathcal{Z}$-polyhedron where some rows of its corresponding affine lattice are interpreted as size parameters. We can always express a $\mathcal{Z}$-polyhedron such that its size parameters have a one-to-one correspondence with certain indices of its coordinate polyhedron.

2.4 Presburger Sets

Presburger (integer) formulae consists of affine inequalities over integer variables and logical operators combining affine inequalities. Formally, a Presburger formula is defined as follows [22]:

$$f = a \mid \exists x.f \mid f \wedge f \mid f \vee f \mid \neg f$$

where $a = t < t$ and $t = 0 \mid 1 \mid x \mid t + t \mid -t$. Here, x represent an integer variable.

The Presburger atom a is an single affine expression over integer variables, which can be represented as a half-space (or trivially an integer polyhedron). The universe, $\mathcal{U}$, is also an integer polyhedron. The family of unions of integer polyhedra is closed under intersection, union and difference. In the absence of the existential qualifier, any f can be expressed as a union of integer polyhedra. However, the union of integer polyhedra is not closed under the existential qualifier, which can be viewed as a projection. It has been proved recently that

Presburger Set Construction Rule	Corresponding operation on a union of $\mathcal{Z}$-polyhedra
Presburger atom $a < a'$	A half space whose constraint is $a < a'$
$f_1 \wedge f_2$	$\mathcal{D}_{f_1} \cap \mathcal{D}_{f_2}$
$f_1 \vee f_2$	$\mathcal{D}_{f_1} \cup \mathcal{D}_{f_2}$
$\neg\ f$	$\mathcal{U} \setminus \mathcal{D}_f$
$\exists x\ .\ f$	image of $\mathcal{D}_f$ by $((z, x) \rightarrow z)$ function

Table 1: Conversion rule from a Presburger set to a union of $\mathcal{Z}$-polyhedra; f, f_1 and f_2 are Presburger formulae and $\mathcal{D}_f$, $\mathcal{D}_{f_1}$ and $\mathcal{D}_{f_2}$ are the corresponding unions of $\mathcal{Z}$-polyhedra

the family of unions of $\mathcal{Z}$-polyhedra is closed under image by arbitrary affine functions, [14]. As a corollary, any Presburger set can always be written as a union of $\mathcal{Z}$-polyhedra. The formal conversion rule is given in Table 1. The existential qualification on variable x can be viewed as the image of the set $\mathcal{D}_f$ satisfying f by the function that eliminates the variable x. Note that a point z exists in the set described by $\exists x.f$ if and only if $(z, x) \in \mathcal{D}_f$. Since the family of unions of $\mathcal{Z}$-polyhedra is closed under intersection, union, difference and image by affine function, the results of any operation in Table 1 is still a union of $\mathcal{Z}$-polyhedra.

To derive the iteration space of loop in example 1, we will first express it as a Presburger set with the introduction of existential variables to handle mod and div operators. Then, using the closure result mentioned above, we obtain an equivalent representation of the iteration space as a union of four $\mathcal{Z}$-polyhedra.

3 Illustrating Example

We first illustrate the problem and our approach with the help of the following example. Our goal is to express explicitly the iteration spaces as unions of $\mathcal{Z}$-polyhedra and exact data dependence relation on these representation so that further analyses and transformations, specially those in the polyhedral model, can be applied.

```
for i=2 to 2N
  C[i]=0;                          --- R
for i=1 to N
```

```
for j=(i div 2) to N step 2
  C[i+j]=C[i+j]+A[i]*B[j];   --- S
```

Here, `div` represents integer division (also denoted by $'/'$). The iteration space $\mathcal{D}$ of a statement is the set of all valid loop indices of its surrounding loops. The iteration space $\mathcal{D}_R$ of statement R is $\{i \mid 2 \leq i \leq 2N\}$. It is well known that the iteration space of a statement in ACLs can be represented by an parameterized integer polyhedron. However, $\mathcal{D}_S$, the iteration space of statement S, cannot be represented by a polyhedron because of integer division on lower bounds and the non-unit stride.

However, we can write $\mathcal{D}_S$ as the following set:

$$\mathcal{D}_S = \{i, j \mid 1 \leq i \leq N; (i \text{ div } 2) \leq j \leq N; (j - (i \text{ div } 2)) \bmod 2 = 0\}$$

Now, we want to derive a Presburger set by introducing new variables and using the division rule. We introduce a new integer variable $\alpha = i$ div 2.

$$\mathcal{D}_S = \{i, j \mid \exists \alpha, 1 \leq i \leq N; \alpha \leq j \leq N; (j - \alpha) \bmod 2 = 0; \alpha = i \text{ div } 2\}$$

By the division rule, we replace $\alpha = i$ div 2 with $0 \leq i - 2\alpha \leq 1$.

$$\mathcal{D}_S = \{i, j \mid \exists \alpha, 1 \leq i \leq N; \alpha \leq j \leq N; (j - \alpha) \bmod 2 = 0; 0 \leq i - 2\alpha \leq 1\}$$

The modular operation $(j - \alpha) \bmod 2 = 0$ can be removed by introducing a new integer variable β and setting $j - \alpha = 2\beta$. Finally, we get the following Presburger set.

$$\mathcal{D}_S = \{i, j \mid \exists \alpha, \beta, 1 \leq i \leq N; \alpha \leq j \leq N; j - \alpha = 2\beta; 0 \leq i - 2\alpha \leq 1\}$$

We can view this Presburger set as the image of $\mathcal{T} = \{i, j, \alpha, \beta \mid 1 \leq i \leq N; \alpha \leq j \leq N; j - \alpha = 2\beta; 0 \leq i - 2\alpha \leq 1\}$ by the projection $(i, j, \alpha, \beta) \rightarrow (i, j)$. Note that $(i, j, \alpha, \beta) \in \mathcal{T}$

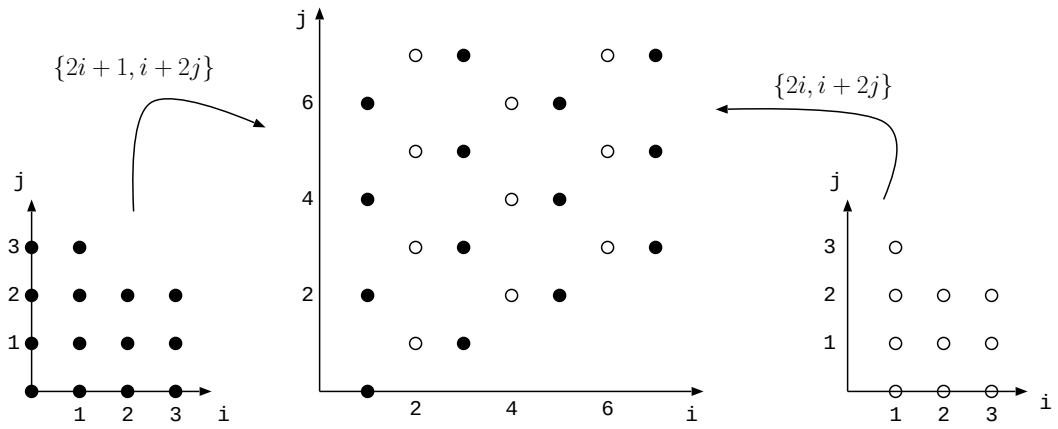

Figure 1: Illustrating Example: The diagram in the center is the iteration space, $\mathcal{D}_S$, of the statement S which is a union of two $\mathcal{Z}$-polyhedra; the leftmost object is the coordinate polyhedron of the black points in $\mathcal{D}_S$; and the rightmost object is the coordinate polyhedron of the white points in $\mathcal{D}_S$.

if and only if $(i, j) \in \mathcal{D}_S$.

In this example, the execution of statement S at the iteration (i', j') requires the value of the array C at $[i' + j']$. We want to find *the* most recent instance $(i, j) \in \mathcal{D}_S$ of S that wrote its result to that location of the array C. Also, the answer (i, j) should be a function of (i', j').

First, we find the set of all instances that write to $C[i' + j']$ before the execution of the (i', j') instance. This set can be described by the following four conditions: (i) $(i, j) \in \mathcal{D}_S$, (ii) $(i', j') \in \mathcal{D}_S$, (iii) $i + j = i' + j'$ and (iv) $i \leq i' - 1$. Now, we express this set as a single polyhedron $\mathcal{P}$ indexed by $(i, j, \alpha, \beta, \alpha', \beta', i', j', N)$ where (i', j', N) are parameters. Once we find the lexicographic maximum as a function of (i', j', N), the first two indices will be the computation that writes the value read by the (i', j') instance. The parameterized lexicographic maximum of $\mathcal{P}$ can be obtained by the PIP solver [7].

Given a valid (i', j') we are assured that (i, j) belongs $\mathcal{D}_S$ by the nature of the constraints posed above. However, we need to devise a mechanism to determine whether (i', j') belongs $\mathcal{D}_S$. To verify whether $(i'j') \in \mathcal{D}_S$, we need to find a point $(i', j', \alpha', \beta') \in \mathcal{T}$. Alternatively, we must explicitly represent the set without the use of extra variables. This iteration space can be represented directly, as a union of $\mathcal{Z}$-polyhedra.

This iteration space $\mathcal{D}_S$ for $N = 7$ is shown in Figure 1. By investigating the figure, we see that a translation from any valid iteration point along the $(2, 1)$ and $(0, 2)$ directions results in either a valid point, or outside the polyhedral closure of the domain. However, note that it is impossible to move a black point to a white point with the $(2, 1)$ and $(0, 2)$ generators. The black and white points correspond to two different affine lattices. The set of black points can be described as follows:

$$\mathcal{B} = \{2y_1 + 1, y_1 + 2y_2 \mid 1 \leq 2y_1 + 1 \leq N; y_1 + 2y_2 \leq N; 0 \leq y_2\} \tag{2}$$

The set of white points can be described as:

$$\mathcal{W} = \{2y_1, y_1 + 2y_2 \mid 1 \leq 2y_1 \leq N; y_1 + 2y_2 \leq N; 0 \leq y_2\} \tag{3}$$

This iteration space can also be computed in a systematic way, i.e., by computing the image of $\mathcal{T}$ by the function $(i, j, \alpha, \beta) \rightarrow (i, j)$. The image of polyhedra by affine functions can be represented as a union of $\mathcal{Z}$-polyhedra. More generally, any arbitrary Presburger set can be expressed by a union of $\mathcal{Z}$-polyhedra.

Finally, what is remaining is to express the lexicographic maximum obtaining by the PIP solver in terms of y_1 and y_2 instead of i' and j' for each of the two $\mathcal{Z}$-polyhedra in the iteration space of statement S. This can be achieved by simply replacing i' by $2y_1 + 1$ and j' by $y_1 + 2y_2$ for $\mathcal{B}$ and i' by $2y_1$ and j' by $y_1 + 2y_2$ for $\mathcal{W}$.

4 Problem Definition and ILP Formulation

In this section, we want to answer the following three questions: (i) for the execution of a statement at a certain iteration (given as a vector of loop indices and size parameters) requiring the value of a memory location, which statement produces the value and at which iteration; (ii) what is the set, $\mathcal{D}$, of valid iteration vectors and its preferred representation;

```
                Input program Grammar

Program     : Stmt*
Stmt        : Loop | If | Assignment
Loop        : for Index = ALExpr to ALExpr step PINT
                 Stmt*
If          : if '(' ABExprs ')'
                 Stmt*
ArrayRef    : Var[ALExpr*]
Assignment  : ArrayRef = func '(' ArrayRef* ')'
ALBExprs    : ALBExprs '&&' ALBExprs |
              ALBExprs '||' ALBExprs |
              exists '(' Index ',' ALBExprs ')' |
              ALBExpr
ALBExpr     : ALExpr Relation ALExpr
Relation    : '>' | '<' | '==' | '=<' | '=>'
ALExpr      : ALExpr '+' Factor |
              ALExpr '-' Factor |
              ALExpr mod PINT |
              ALExpr div PINT |
              max '(' ALExpr* ')' |
              min '(' ALExpr* ')'
Factor      : PINT Index | Index | PINT
```

Figure 2: Input program grammar. `Index` is an identifier; `ALExpr` (called affine lattice expressions) are functions of outer indices and size parameters; `PINT` is a positive integer; `ArrayRef` is a reference to a multidimensional array that is accessed by an affine lattice expression of outer indices and size parameters; `func is` a strict, side-effect free function; `div` is a integer division (also denoted by $'/'$); and `mod` is a modular operation (also denoted by $'\%'$).

and (*iii*) how can we represent the iteration vector of the producer in concordance with the representation of $\mathcal{D}$.

4.1 Input programs

We first describe the class of programs that can be analyzed by our technique. This class contains ACLs but, in addition, allows loops with non-unit stride and arbitrarily nested mod, integer division, max and min operations on surrounding indices in loop bounds, guards and memory access functions. In addition, we also allow existential quantifiers in guards. The grammar for our input programs is given in Figure 2.

We now present three example programs: a contrived example to illustrate the ILP formulation, a well-known matrix-matrix multiplication algorithm on a $2D$-torus of processors,

and a stencil program.

Example 2 *Contrived example for* ILP *formulation:*

```
for i=N%5 to N step 2
  for j=i/2+i%4 to N+i%3
    X[j] = X[i/2+j%2];
```

Example 3 *Cannon's algorithm for multiplying two matrices. This example shows what kinds of programs can be described by the grammar in Figure 2. However, it is not useful for the rest of the paper because of trivial data dependence except dependence on input arrays.*

```
for k = 0 to 5
  forall i = 0 to 5
    forall j = 0 to 5
      C[i,j]=C[i,j]+A[i,(i+j+k)%6]*B[(i+j+k)%6];
```

For details please refer to [5].

Example 4 *Stencil computation (*RED-BLACK SOR*):*

```
for t = 0 to 2M
  for i = 1 to N-1
    for j = (t+i-1)%2+1 to N-1 step 2
      X[i,j]=(X[i-1,j]+X[i+1,j]+X[i,j-1]+X[i,j+1])/4;
```

In the paper, we use "statement" and "assignment" interchangeably and always mean assignment in terms of grammar. However, we distinguish an assignment and its operation (or instance) which is a distinct execution of the statement. A statement can be executed multiple times, but an operation is unique in the entire execution of a program. An operation is described completely by a statement and its associated iteration vector. We denote an operation $z \in \mathcal{D}_S$ of a statement S by (S, z).

expression	Additional variable	Equivalent form	Additional Constraint
`ALExpr` div c	$\exists\alpha \in \mathbb{Z}$	α	$(0 \leq$ `ALExpr` $- c\alpha \leq c-1)$
`ALExpr` mod c	$\exists\alpha \in \mathbb{Z}$	`ALExpr` $- c\alpha$	$(0 \leq$ `ALExpr` $- c\alpha \leq c-1)$

Table 2: Equivalent form for `ALExpr`$\odot c$ where $\odot$ is an integer division or mod by the positive integer c.

4.2 Iteration Space as a Presburger Set

We will first explain how to express the iteration space of input loop nests as Presburger sets. A Presburger set can be always seen as an image of a union of higher dimensional polyhedra by the projection that eliminates all unnecessary variables. We use the higher dimensional polyhedron for our ILP formulation. Its canonical projection along all the existential variables yields the iteration space as a Presburger set.

First, let us consider the simple case where the expression is a form of $\alpha = AE \odot c$ where $\odot$ is either an integer division or modular operation and c is a positive integer e.g., $\alpha = (i+j)$ div 2. By the division rule, we may replace it by $0 \leq i + j - c\alpha \leq c - 1$. For the general case, transformations are given in Table 2. In obtaining the equivalent form, we have introduced an existential variable and an additional constraint.

The procedure to construct a Presburger set from arbitrary loop bounds is to introduce a new existential variable for every mod or div operator until all the conditions are affine expressions (with nested max and min). The maximum or minimum of affine expressions can be transformed into disjunction and conjunction of affine expressions. Our iteration space is then the projection of a union of polyhedra along the canonic directions corresponding to all the introduced existential variables.

We illustrate this algorithm through its application to Example 2. The iteration space can be expressed by

$$\{i, j \mid N\%5 \leq i \leq N; (i - N\%5)\%2 = 0; (i/2) + i\%4 \leq j \leq N + i\%3\}$$

Now, we replace $N\%5$ by $N - 5\alpha_1$ by introducing a new variable α_1 and the new constraint

$0 \leq N - 5\alpha_1 \leq 4$. Then, we get

$$\{i, j \mid \exists \alpha_1, 0 \leq N-5\alpha_1 \leq 4; N-5\alpha_1 \leq i \leq N; (i-(N-5\alpha_1))\%2 = 0; (i/2)+i\%4 \leq j \leq N+i\%3\}$$

We introduce three more variables α_2, α_3 and α_4 to replace $i\%4$, $i\%3$ and $(i-(N-5\alpha_1))\%2$ by $i - 4\alpha_2$, $i - 3\alpha_3$ and $(i - (N - 5\alpha_1)) - 2\alpha_4$, respectively along with the introduction of the constraints $0 \leq i - 4\alpha_2 \leq 3$, $0 \leq i - 3\alpha_3 \leq 2$ and $0 \leq (i - (N - 5\alpha_1)) - 2\alpha_4 \leq 1$.

$$\begin{aligned}\{i, j \mid \exists \alpha_1, \alpha_2, \alpha_3, \alpha_4, 0 \leq N - 5\alpha_1 \leq 4; 0 \leq i - 4\alpha_2 \leq 3; 0 \leq i - 3\alpha_3 \leq 2; 0 \leq (i - (N - 5\alpha_1)) - 2\alpha_4 \leq 1;\\ N - 5\alpha_1 \leq i \leq N; 0 \leq (i - (N - 5\alpha_1)) - 2\alpha_4 \leq 1; (i/2) + (i - 4\alpha_2) \leq j \leq N + (i - 3\alpha_3)\}\end{aligned}$$

To remove the only div (or $'/'$) operator, we introduce α_5 to replace $(i/2)$ by α_5 along with the introduction of the constraint $0 \leq i - 2\alpha_5 \leq 1$.

$$\begin{aligned}\{i, j \mid \exists \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5,\\ 0 \leq N - 5\alpha_1 \leq 4; 0 \leq i - 4\alpha_2 \leq 3; 0 \leq i - 3\alpha_3 \leq 2; 0 \leq (i - (N - 5\alpha_1)) - 2\alpha_4 \leq 1; 0 \leq i - 2\alpha_5 \leq 1;\\ N - 5\alpha_1 \leq i \leq N; 0 \leq (i - (N - 5\alpha_1)) - 2\alpha_4 \leq 1; \alpha_5 + (i - 4\alpha_2) \leq j \leq N + (i - 3\alpha_3)\}\end{aligned}$$

Note that the constraints consists of only affine constraints. This Presburger set is equivalent to the projection of the polyhedron, $\mathcal{P}_{\text{contrived}}$:

$$\begin{aligned}\{i, j, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5 \mid\\ 0 \leq N - 5\alpha_1 \leq 4; 0 \leq i - 4\alpha_2 \leq 3; 0 \leq i - 3\alpha_3 \leq 2; 0 \leq (i - (N - 5\alpha_1)) - 2\alpha_4 \leq 1; 0 \leq i - 2\alpha_5 \leq 1;\\ N - 5\alpha_1 \leq i \leq N; 0 \leq (i - (N - 5\alpha_1)) - 2\alpha_4 \leq 1; \alpha_5 + (i - 4\alpha_2) \leq j \leq N + (i - 3\alpha_3)\}\end{aligned}$$

by the function $(i, j, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5 \rightarrow i, j)$.

The iteration space in Example 4 can be constructed by introducing two additional

variables to obtain the Presburger set, $\mathcal{D}_{\text{stencil}}$

$$\{t, i, j | \exists \alpha_1, \alpha_2, 0 \leq t \leq 2M; 0 \leq i \leq N-1;$$
$$0 \leq t+i-1-2\alpha_1 \leq 1; (t+i-2\alpha_1) \leq j \leq N-1; j-(t+i-2\alpha_1)-2\alpha_2 = 0\}$$

In general, we may have guards on the execution of statements. This simply requires the addition of the constraints in the `if` condition to the iteration space of the corresponding statement. A subtle point to note is that user-defined existential variables may only occur in the `if` statements.

4.3 Predicate for execution order

This section focuses on the ordering of operations in input programs. In other words, it provides the condition that an operation (S, z') is executed before another operation (R, z). We denote the depth of common loops of S and R by N_{SR} and the textual order of statements S and R by the boolean predicate B_{SR}.

A operation (S, z') is executed before (R, z) if

$$z'[1 \ldots N_{SR}] \prec z[1 \ldots N_{SR}] \vee (B_{SR} \wedge z'[1 \ldots N_{SR}] = z[1 \ldots N_{SR}]) \tag{4}$$

When $N_{SR} = 0$, the condition will be just B_{SR}.

For our illustrating example, $C_{SR} = 0$ and B_{RS} is true. So, an operation of the assignment R precedes any operations of S. An operation (R, z) precedes (R, z') if $z \prec z'$. Similarly, (S, z) precedes (S, z') if $z \prec z'$.

Note that this sequencing predicate is the same as the predicate for ACLs. It is because steps in input programs are positive integers and the conditional statements do not affect the order of iteration vectors. A conditional statement only restricts the space of valid iteration vectors. For a more detail description, please refer to Feautrier's work [8] on the exact dependence analysis of ACLs.

4.4 ILP formulation

We have statements whose iteration spaces are Presburger sets. For a consumer statement C whose memory (read) access function is M_C and a candidate producer statement P whose (write) access function is M_P.

We will assume that the iteration space of a statement given by the Presburger set $\mathcal{D}$ is such that it is the image of a *single* polyhedron $\mathcal{P}$ by the canonical projection along the directions corresponding to existential variables. This is restrictive. However, we will see that generalizations are straightforward extensions to the basic technique presented for this case.

First, we want to find all the instances of P that write to the location $M_C(z')$ before z' where $z' \in \mathcal{D}_C$. This set $\mathcal{C}$ can be described as:

$$\mathcal{C} = \{z \in \mathcal{D}_P \mid z' \in \mathcal{D}_C; M_P(z) = M_C(z'); z_c \prec z'_c\}$$

where z_c (resp., z'_c) is $z[1 \ldots N_{PC}]$ (resp., $z'[1 \ldots N_{PC}]$). Note that depending on textual order z_c may be allowed to be z'_c. Using the fact that $z \in \mathcal{D}_P$ iff $(z, \alpha) \in \mathcal{P}_P$ we get

$$\mathcal{C}' = \{(z, \alpha) \in \mathcal{P}_p \mid (z', \alpha') \in \mathcal{P}_C; M_P(z) = M_C(z'); z_c \prec z'_c\}$$

As a special case, if the memory function M_P and M_C are both affine functions, we have the same ILP formulation as that of ACLs. However, its naive formulation causes the problem that the ILP solution depends on z', as well as α'. We can avoid this by treating α' as unknown indices as well, rather than parameters, when constructing the parameterized polyhedra $\mathcal{Q}$ given as

$$\mathcal{Q} = \{(z', z, \alpha, \alpha') \mid (z', \alpha') \in \mathcal{P}_C; (z, \alpha) \in \mathcal{P}_C; M_P(z) = M_C(z'); z_c \prec z'_c\}$$

$\mathcal{Q}$ is parameterized by only z' and size parameters. In the solution of the ILP solver,

(z, α, α') will be expressed as a function of z' and size parameters. The intuition behind such a formulation is that the values of (α, α') do not matter as long as (α, α') simply exists and z is the corresponding lexicographic maximum.

Consider the stencil computation in Example 4. We want to formulate an ILP problem to find out the producer of (t', i', j') as a function of the indices t', i' and j' for the memory reference $X[i, j-1]$. Then, the set of candidate operations can be specified by

$$\mathcal{Q}_{\text{stencil}} = \{t', i', j', t, i, j, \alpha_1, \alpha_2, \alpha'_1, \alpha'_2 \mid (t', i', j') \in \mathcal{P}_{\text{stencil}}; (t, i, j) \in \mathcal{P}_{\text{stencil}}; i' = i; j'-1 = j; (t, i, j) \prec (t$$

This set is a polyhedra parameterized by (t', i', j'). Using the PIP solver, we find the last-write operation $(t, i, j, \alpha_1, \alpha_2, \alpha_3, \alpha'_1, \alpha'_2, \alpha'_3)$ as a function of (t', i', j'). It is only the first three components, (t, i, j), of the last-write operation of (t', i', j') that we seek. Note that the proper values for the remaining variables exist given (t', i', j') and (t, i, j).

In fact, $\mathcal{Q}_{\text{stencil}}$ is not a single polyhedron because of the lexicographic order, but a union of three polyhedra. The lexicographic order $(t, i, j) \prec (t', i', j')$ is decomposed to three cases: $t < t'$, $t = t' \wedge i < i'$ and $t = t' \wedge i = i' \wedge j < j'$. So, for each polyhedron, a different ILP problem is solved and all the individual solutions are combined to obtain the producer iteration that last wrote the value. In the presence of multiple candidate producer statements these solutions are further combined to obtain the producer statement and its iteration. For a detailed description of such steps, again refer to Feautrier's work [8].

With this understanding, it is straightforward to see that when the iteration space of a statement is the image of a union of polyhedra, we can solve the ILP problem once for each pair of consumer and producer polyhedron in the unions and subsequently combine the results to obtain the required producer iteration. Note that the number of calls to PIP are a function of the number of the disjunctions in the Presburger set, not that of the exponential number of elements in the union of $\mathcal{Z}$-polyhedra.

Our memory access functions are not always simple affine functions (refer to Example

2). To obtain the required Presburger set for complex memory access, we simplify the constraints $M_p(z) = M_C(z')$ using the techniques presented previously in the construction of the iteration space of statements.

With the help of Example 2, we now illustrate this approach. Consider the set that satisfies $i'/2 + j'\%2 = j$:

$$\{(i', j', i, j) \mid i'/2 + j'\%2 = j\}$$

We introduce two existential variables β_1 and β_2 to remove the modular the integer division operations to obtain

$$\mathcal{M} = \{i', j', i, j, \beta_1, \beta_2 \mid 0 \leq i' - 2\beta_1 \leq 1; 0 \leq j' - 2\beta_2 \leq 1; \beta_1 + (j' - 2\beta_2) = j\}$$

Finally, the set of candidate operations can be specified by

$$\mathcal{Q}_{\text{contrived}} = \{(z', z, \alpha, \alpha', \beta) \mid (z', \alpha') \in \mathcal{P}_{\text{contrived}}; (z, \alpha) \in \mathcal{P}_{\text{contrived}}; (z', z, \beta) \in \mathcal{M}; z \prec z'\}$$

Once again, we treat this set as a polyhedra parameterized by z' and size parameters and find the lexicographic maximum using the PIP solver to obtain the producer for the iteration at z'.

4.5 Iteration Space as a Union of $\mathcal{Z}$-polyhedra

Thus far, we have expressed the iteration space of statements as Presburger sets and presented in detail how to construct these sets. However, the existential quantification in Presburger equations makes it difficult to obtain the precise set of iterations. For this, we express Presburger sets as a union of $\mathcal{Z}$-polyhedra and specialize the last-write function to each individual $\mathcal{Z}$-polyhedron. Let $\mathcal{ZP} = \{Ly + l | y \in \mathcal{P}\}$ be a $\mathcal{Z}$-polyhedron in the union and $f(z')$ is the last-write function. The last-write function in terms of the coordinates of the $\mathcal{Z}$-polyhedron is then simply $f(Ly + l)$.

5 Conclusion & Future Work

We presented a compiler analysis for performing exact value-based dependence analysis for a richer class of "static control loop programs." This class is significantly more general than the largest previously known class, namely *affine control loops* for which such analysis was possible [8, 25, 24]. It includes loop programs whose iteration spaces are arbitrary Pressburger sets and whose access functions are affine functions extended with `div` and `mod` operators and existential quantifiers. Such a class is not only more general, but it is also important, since many common applications such as red-black SOR, and Canon's algorithm for matrix multiplication, can be concisely and naturally written using the extended syntax that we provide.

Our analysis may be viewed as (i) an initial pre-processing step to transform the input specification so that all existentially quantified formulae are replaced by polyhedral sets of higher dimensions; (ii) formulation of the precedence constraints as a parametric integer linear program that can be solved by well known tools such as PIP [7]; and (iii) a final post-processing step that converts the results of this into system of affine recurrence equations defined over $\mathcal{Z}$-polyhedral domains (called $\mathcal{Z}$-Polyhedral SAREs, or ZPSAREs). Because of the precisely defined pre- and post-processing steps, we are able to simply reuse existing machinery and tools that have been previously developed for the limited case of ACLS.

The output of the analysis, ZPSAREs, can exploit a novel representation of $\mathcal{Z}$-polyhedra [14] and can be analyzed for parallelism using sophisticated and more precise scheduling techniques, that allow us to detect more parallelism than is possible under the polyhedral model alone [13].

Our ongoing work involves the implementation of these techniques. The construction of the precise iteration space requires two important tools: a library for manipulating unions of $\mathcal{Z}$-polyhedra and a component for converting a Presburger set to a union of $\mathcal{Z}$-polyhedra.

References

[1] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE PACT*, pages 7–16, 2004.

[2] A. Darte and Y. Robert. Affine-by statement scheduling of uniform and affine loop nests over parametric domains. *Journal of Parallel and Distributed Computing*, 29(1):43–59, February 1995.

[3] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization.* Birkhäuser, 2000.

[4] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, October 2005.

[5] Alain Darte, Michèle Dion, and Yves Robert. A characterization of one-to-one modular mappings. *Parallel Processing Letters*, 5:145–157, 1996.

[6] E. De Greef, F. Catthoor, and H. De Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. In *Parallel Processing and Multimedia*, Geneva, Switzerland, July 1997.

[7] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationelle*, 22(3):243–268, Sep 1988.

[8] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, Feb 1991.

[9] P. Feautrier. Some efficient solutions to the affine scheduling problem, Part I, one-dimensional time. Technical Report 28, Labaratoire MASI, Institut Blaise Pascal, April 1992.

[10] P. Feautrier. Some efficient solutions to the affine scheduling problem, Part II, multidimensional time. Technical Report 78, Labaratoire MASI, Institut Blaise Pascal, October 1992.

[11] Agustin Fernández, José M. Llabería, and Miguel Valero-García. Loop transformation using nonunimodular matrices. *IEEE Trans. Parallel Distrib. Syst.*, 6(8):832–840, 1995.

[12] Gautam and S. Rajopadhye. Simplifying reductions. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 30–41, New York, NY, USA, 2006. ACM Press.

[13] Gautam Gupta, DaeGon Kim, and Sanjay V. Rajopadhye. Scheduling in the Z-polyhedral model. In *IPDPS*, pages 1–10, 2007.

[14] Gautam Gupta and Sanjay V. Rajopadhye. The Z-polyhedral model. In *PPOPP*, pages 237–248, 2007.

[15] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, 1967.

[16] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, pages 83–93, February 1974.

[17] V. Lefebvre and P. Feautrier. Optimizing storage size for static control programs in automatic parallelizers. In Lengauer, Griebl, and Gorlatch, editors, *Euro-Par'97*, volume 1300. Springer-Verlag, 1997.

[18] Patrick Lenders and Sanjay Rajopadhye. Multirate vlsi arrays and their synthesis. *IEEE Trans. Comput.*, 46(5):515–529, 1997.

[19] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. *Int. J. Parallel Program.*, 22(2):183–205, 1994.

[20] Amy W. Lim, Shih-Wei Liao, and Monica S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 103–112, New York, NY, USA, 2001. ACM Press.

[21] B. Lisper. Linear programming methods for minimizing execution time of indexed computations. In *Int. Workshop on Compilers for Parallel Computers*, 1990.

[22] Josh MacDonald. Program analysis with presburger integer formulae.

[23] C. Mauras, P. Quinton, S. Rajopadhye, and Y. Saouter. Scheduling affine parametrized recurrences by means of variable dependent timing functions. In *International Conference on Application Specific Array Processing*, pages 100–110, 1990.

[24] D. Maydan, S. P. Amarsinghe, and M. Lam. Array data flow analysis and its use in array privatization. In *Principles of Programming Languages*, pages 2–15. ACM, January 1993.

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

[26] Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Trans. Program. Lang. Syst.*, 22(5):773–815, 2000.

[27] Patrice Quinton and Vincent Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1(2):95–113, 1989.

[28] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In *Proceedings, Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 488–503, New Delhi, India, December 1986. Springer Verlag, LNCS 241. Later appeared in Parallel Computing, June 1990.

[29] J. Ramanujam. Beyond unimodular transformations. *J. Supercomput.*, 9(4):365–389, 1995.

[30] Sailesh Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays*. PhD thesis, Stanford University, Information Systems Lab., Stanford, Ca, October 1985.

[31] W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe. A unified framework for schedule and storage optimization. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 232–242. ACM Press, 2001.

[32] Jingling Xue. Automating non-unimodular loop transformations for massive parallelism. *Parallel Computing*, 20(5):711–728, 1994.