Dependence Analysis and Parallelizing Transformations

S. Rajopadhye[*]

1 Introduction

The first impression one gets when phrases like *dependence analysis* and *automatic parallelization* are mentioned, is that of loop programs and array variables. This is not surprising, since loops is the classic repetitive structure in any programming language, and clearly, this is where programs spend a significant amount of their time. Second, because of the early impetus on high performance computing for large numerical applications (eg. FORTRAN programs) on supercomputers, there has been a long research effort on parallelizing such programs. The area has been active for over a quarter century, and a number of well known texts on this topic are readily available.

Any approach to automatic parallelization must be based on the following fundamental notions: (i) detection of which statements in which iterations of a (possibly multiply nested and possibly imperfect) loop *depend* on each other, in the precise sense that one of them produces a value that is consumed by the other; (ii) hence determining which operations can be executed in parallel; and (iii) transforming the program so that this choice of parallelization rendered explicit. The first problem is called dependence analysis, the second constitutes the additional analysis necessary to *choose* the parallelization, and the third is called program or restructuring.

In all generality, these are extremely difficult problems. Nevertheless, for certain classes of programs elegant and powerful methods are available. Therefore, rather than giving "yet another survey" of a vast field, we present in this chapter, a somewhat less known approach,

[*]Email: `Sanjay.Rajopadhye@colostate.edu`; Computer Science Department, Colorado State University, Fort Collins CO 80523

called the *polyhedral model.* The model is based on a *mathematical* model of both the program to be parallelized and the analysis and transformation techniques used. It draws from operational research, linear and non-linear programming and functional and equational programming. It's applicability is however restricted to a well defined and limited (but nevertheless important) subset of imperative programs.

Essentially, programs in the polyhedral model are those for which the above three questions can be resolved systematically and optimally. This class is called *affine control loops* or ACL's. This restriction is motivated by two principal reasons.

- ACL's constitute an important (sub) class of programs. Many programs spend the majority of their time in such loops. This occurs not merely in numerically intensive computations (where such loops are more or less *siné qua non*) but also in other domains such as signal and image processing, multimedia applications, etc.

- There exist elegant and powerful mathematical techniques for reasoning about and transforming such programs. Drawing from the methods of operational research and linear and non-linear programming, they yield provably optimal implementations for a variety of cost criteria.

Independent of *how* the parallelism is to be specified (notions of a language for *expressing* the parallelized program, its semantics, and its efficient *implementation* on a parallel machine), there are two primary hurdles to detecting parallelism in a sequential (imperative) program. The first is the re-utilization of memory—the problem of false (anti- and output-) dependences. The fact that the same memory location is used to store two distinct values, implies that all computations that use the first value must be executed before it is (over) written by the second regardless of whether or not they are independent of the latter (and hence potentially parallelizable). Indeed, the *dependence analysis* step that precedes any parallelization consists essentially of identifying the "true" dependences between the different operations of the program.

The second aspect of imperative sequential languages that hinder parallelism detection is what is called the *serialization* of *reductions.* Reductions are operations that combine many (say, n) data values using an associative (and often commutative) operator. In an imperative sequential program such computations are serialized (often in an arbitrarily chosen manner), merely because the programming language cannot express the independence of the result on the order of evaluation. The dependence graph obtained naively from such a program is a sequence of length n, and does not allow any parallelism. However, the associativity allows us to potentially execute the computation in logarithmic time (on $n/2$ processors with only binary operators), or in time $\frac{n}{p} + \lg p$ on p processors, which is of the order $\Theta(\frac{n}{p})$ if $n > p \lg p$ (such a parallel implementation is thus work-optimal).

The polyhedral model enables the resolution of these two problems. First of all, it allows for an *exact data-flow analysis* of ACL's which completely eliminates false dependences, thus enabling essentially "perfect" dependence analysis. The result of such an analysis yields an intermediate form of the program that may be formally described as a *System of Affine Recurrence Equations* (SARE) whose variables are defined over *polyhedral domains*. The polyhedral model enables a second analysis of such SARE's in order to detect the presence of reductions in the program.

It is worth noting here that the formalism of SARE's is interesting in and of itself, rather than being a "mere" intermediate form for the analysis of ACL's. It constitutes a class of equational programs that are usually closer to the mathematical notions that underlie the algorithms embodied in most ACL's. The notation of SARE's can be extended to express reduction operations, and this yields an even more intuitive and mathematical formalism useful for algorithm designers. Because they are equational, program development in this formalism can benefit formal methods such as program verification, synthesis by correctness-preserving transformations, abstract interpretation, etc. The ALPHA language developed at Irisa subscribes to this view, and the tools and techniques used by the MMALPHA system for parallelizing ALPHA programs are based fundamentally on the same techniques that we describe in this chapter for ACL's.

The remainder of this chapter is organized as follows. In Section 2 we first present an overview of the parallelization process. Next, Section 3 describes the mathematical foundations that we will need. In Section 4 we describe the algorithms for exact data-flow analysis of ACL's. We digress in Section 5 to show how the notation of SARE's together with a mechanism for describing reduction operations can constitute a full fledged equational, data-parallel language (this section may be skipped at first reading). Next, Sections 6 through 8 respectively describe the methods of static analysis of SARE's and treat three important problems, namely that of scheduling an SARE, and that of allocating the computations to processors and to memory locations. Section 9 then describes the problem of generating code from a scheduled and appropriately transformed SARE. Finally we describe limitations, open problems and conclude with some bibliographic notes.

2 Overview of the Parallelization Process

Deriving a parallel program in the polyhedral model consists of the following steps (note that the steps may not all be performed in the order stated, nor is it essential that *all* steps be performed).

1. We analyse the sequential program (i.e., the ACL) in order to determine the corresponding polyhedral SARE.

2. We perform a second analysis pass on the SARE to detect reductions and other collective associative operations. Due to space constraints we do not give details of this analysis here.

3. Now, a polyhedral SARE explicitly names *every value* computed by (i.e., the result of *every operation* in) the program. Hence our next step consists of determining three functions related to its final implementation. For each operation of the SARE, we determine:

    - an *execution date*, i.e., a schedule;
    - a *processor allocation* which specifies the processor where that operation is to be executed; and
    - a *memory address* where the result will be stored.

    The schedule may impose an order on the "accumulations" of the intermediate results in reductions and scans. Vis-à-vis the other two functions, one often uses the convention implied by the "owner-computes" rule, namely that a value computed by any processor is stored in the (local) memory of that processor itself. In this case, a memory allocation function defined as a pair—specifying a processor, and a (local) address on this processor—is sufficient to subsume the processor allocation function.

Note that there are two issues related to the choice of these functions. First of all, they must be *valid*, according to certain architectural and semantic constraints. In addition, it is desirable that they be chosen so as to optimize certain cost criteria. We will see that the polyhedral model provides us with elegant techniques to address both these issues.

4. Once these functions are chosen, we construct an equivalent SARE that respects the following conventions: certain indices of *all* the variables of the SARE, called the "temporal" indices, represent time, i.e., the schedule that we have chosen, others represent the processor indices, and still others represent memory addresses.

5. From this transformed SARE, we produce code that (i) respects the order implied by the temporal indices, (ii) has the parallelism implied by the processor indices, and (iii) allocates and accesses the memory as specified by the memory indices. This code is in the form of an ACL which is now parallelized and optimized for various cost criteria (that guided the choice of the three functions in step 3 above)

3 Notations and mathematical foundations

In an affine control loop (ACL) program, there are two distinct classes of variables: *data* variables, and *index variables* (the latter set also includes certain variables called *size parameters* which are assigned a single value of any instance of the program). Data variables are *multidimensional arrays* (scalars being viewed as zero-dimensional arrays), and represent the primary "values" computed by the program. Index variables are integer typed, are never explicitly assigned (they get their values implicitly in loops), and are used to "access" data variables.

The only control construct (other than sequential statement composition) is the `for` loop (note that there is no conditional *if-then-else* construct[1]). The lower (cf. upper) bound of such a loop is an *affine* expression of the surrounding loop indices and the size parameters, or the maximum (cf. minimum) of such expressions. The body of the loop is either

[1]This constraint may be relaxed without any loss of generality, provided that the conditionals are (conjunctions of) affine inequalities involving only the index variables and size parameters.

- another loop;
- an assignment statement; or
- a sequential composition of any of the above

In any assignment statement, the left hand side (lhs) is a data variable, and the right hand side (rhs) is an expression involving data variables. The access function of the data variables, whether on the lhs or the rhs, is an affine function of the surrounding loop indices and the size parameters. For our later analysis, we shall assume that the rhs expression is atomic.

In an ACL, an assignment statement S, is executed many times for different values of the surrounding loop indices. The loop indices are said to be *valid* if they are within the appropriate bounds, and the set of valid indices surrounding S is called its *iteration domain*, D. Since there are no conditionals, each loop body *must* be executed exactly once for *each* valid value of the surrounding indices. Every *operation* in the program is therefore uniquely identified by $\langle S_i, z\rangle$, where S_i is a statement, and $z \in D$ is an integer vector, the *iteration vector*. For two operations O_1 and O_2, we denote by $O_1 \lhd O_2$, the fact that operation O_1 ***precedes*** O_2 ***in the sequential execution order*** of the ACL.

We now recall certain standard definitions regarding polyhedra.

Definition 1. *A* **rational polyhedron** *is a set of the form* $\{z \in \mathcal{Q}^n \mid Qz \geq q\}$*, where* Q *(cf.* q*) is an integer matrix (cf. vector). An* **integral polyhedron** *or a* **polyhedral domain** *(or simply a polyhedron, when it is clear from the context) is the set of* integer *points in a rational polyhedron:*

$$D \equiv \{z \in \mathcal{Z}^n \mid Qz \geq q\} \tag{1}$$

A **parametric family** *of polyhedra corresponds to the case where the* q *is a (vector valued) affine function of* m size parameters p*, i.e.,* $\{z \in \mathcal{Z}^n \mid Qz \geq q - Pp\}$*, or equivalently*

$$\left\{ \begin{pmatrix} z \\ p \end{pmatrix} \in \mathcal{Z}^{n+m} \mid \begin{bmatrix} Q & P \end{bmatrix} \begin{pmatrix} z \\ p \end{pmatrix} \geq q \right\}.$$

We may therefore view such a parametric family of polyhedra as a single $n+m$ *dimensional polyhedron. Note however, that the converse view—namely a single* $n+m$ *dimensional poly-*

hedron being equivalent to an n-dimensional polyhedron, parameterized by m parameters—is valid for rational polyhedra, but not for integer polyhedra. For example, the projection of an integral polyhedron on one of its axes is not necessarily an integral polyhedron. Nevertheless, for reasons of clarity we will often abuse the notation and view an integer polyhedron as a parameterized family of its constituent projections.

Parametric polyhedra also admit an equivalent dual definition in terms of generators *(their* vertices*,* rays *and* lines*) which are all* piece-wise affine *functions of their parameters.*

Example 1: Consider $\{i, j, n \mid 0 \leq j \leq i; j \leq n; 1 \leq n\}$ as a 3-dimensional polyhedron. It has two vertices: $[0, 0, 1]$ and $[0, 1, 1]$. Its rays are the vectors in the set $\{[1, 0, 0], [0, 0, 1], [0, 1, 1]\}$. The same polyhedron, viewed as a 2-dimensional polyhedron (parameterized by n) has two vertices, $\{[0, 0], [n, n]\}$ and a ray, $[1, 0]$.

Example 2: Consider $\mathcal{P}_1 = \{i, j, n, m \mid 0 \leq j \leq i \leq n; j \leq m; 1 \leq n, m\}$ as a 2-dimensional polyhedron (with parameters n and m). Depending on the relative values of its parameters, it is either a triangle (if $n \leq m$) or a trapezium (otherwise). It does not have rays, and we can see that the set of its vertices is a *piece-wise affine* function of its parameters:

$$\begin{cases} \{1 \leq n \leq m\} & \Rightarrow \quad \{[0, 0], [n, 0], [n, n]\} \\ \{1 \leq m < n\} & \Rightarrow \quad \{[0, 0], [n, 0], [n, m], [m, m]\} \end{cases}$$

Finally, the polyhedron $\mathcal{P}_2 = \{i, j, n, m \mid 0 \leq i = j - 1 \leq n\}$ is a line segment (parameterized by n), but in a two-dimensional space. its vertices are $S_2 = \{[0, 1], [n, n + 1]\}$.

We denote by $z_1 \prec z_2$, the (strict) ***lexicographic order*** between two vectors, z_1 and z_2. Note that this is a *total order*, and that the vectors may even be of different dimensions (just like the words in a dictionary). We define $\prec_n$ as the order, $\prec$ applied only to the first n components of two vectors, i.e., if z_1' and z_2' are respectively the first n components of z_1 and z_2, then $z_1 \prec_n z_2$ if and only if $z_1' \prec z_2'$. The inverse order $\succ$, and the related (partial) orders $\preceq$ and $\succeq$ are natural extensions and may be defined analogously.

We may easily see that the ***execution order*** of operations in a sequential ACL (denoted by $\lhd$) is closely related (but not identical) to the lexicographic order. For any two operations, $\langle S_1, z_1 \rangle$ and $\langle S_2, z_2 \rangle$, let n_{12} be the number of common loop indices surrounding S_1 and S_2.

Then

$$\langle S_1, z_1\rangle \lhd \langle S_2, z_2\rangle \equiv \begin{cases} z_1 \preceq_{n_{12}} z_2 & \text{if } S_1 \text{ appears } \textbf{before } S_2 \text{ in the text of the ACL} \\ z_1 \prec_{n_{12}} z_2 & \text{otherwise} \end{cases}$$

Lmax (cf. Lmin) denotes the lexicographic maximum (cf. minimum) of two or more vectors. We remark that the Lmax (or the Lmin) of all the points in a polytope (i.e., a bounded polyhedron) must be one of its vertices, and that the Lmax (cf. Lmin) of two or more piece-wise affine functions is itself a piece-wise affine function. hence the Lmax (cf. Lmin) of all the points in a union of polyhedra is a piece-wise affine function of its parameters.

Definition 2. *A* **recurrence equation** *(*RE*) is an equation of the following form that defines a variable X at all points z, in a domain, D^X*

$$X(z) = D^X \;:\; g(\ldots X(f(z))\ldots) \tag{2}$$

where

- *z is an n-dimensional **index variable**.*
- *X is a **data variable**, denoting a function of n integer arguments; it is said to be an n-dimensional variable.*
- *$f(z)$ is a **dependence function**, $f : \mathcal{Z}^n \rightarrow \mathcal{Z}^n$ it signifies the fact that in order to compute the value of X at z, we need the valued of X at the index pint $f(z)$;*
- *g is an elementary computation (in our later analyses we assume that it is strict and executes in unit time);*
- *the "$\ldots$" indicate that there may be other similar arguments to g, with possibly different dependences.*
- *D^X is a set of points in $\mathcal{Z}^n$ and is called the **domain** of the equation. Often, the domains are **parameterized** with one or more (say, l) size parameters. In this case, we represent the parameter as a vector, $p \in \mathcal{Z}^l$, and use p as an additional superscript on D.*

We may also have multiple equations that define X as follows:

$$X(z) = \begin{cases} & \vdots \\ D_i & : \quad g_i(\ldots X(f(z))\ldots) \\ & \vdots \end{cases} \tag{3}$$

Here, each row of the definition is called a clause *(or* branch*) of the equation, and the domain of X is the (disjoint) union of the domains of each of the clauses $D^X = \bigcup_i D_i$. One may also define an extension of the formalism of* RE*'s that allows one to specify computations that have* reduction operations *involving associative and/or commutative operators, but we this is beyond the scope of this paper.*

An RE is called an ***affine recurrence equation*** (ARE) if each dependence function is of the form, $f(z) = Az + Bp + a$, where A (cf. B) is a $n \times n$ (cf. $n \times l$) matrix, and a is an n-vector.

Definition 3. *A* **system** *or recurrence equations (*SRE*) is a set of mutually recursive recurrence equations, each one defining one of m variables $X_1 \ldots X_m$ over an appropriate domain (D^{X_i} of dimension n_i for X_i). Since the equations are mutually recursive, the dependence function associated with an occurrence of, say X_j, on the rhs of an equation defining X_i is a mapping from $\mathcal{Z}^{n_i}$ to $\mathcal{Z}^{n_j}$. In a system of* ARE*'s (i.e., an* SARE*) the dependence functions are all affine (the A matrices are not necessarily square).*

We often desire to manipulate (i.e., "geometrically transform") the domains of variables in an SRE and construct an equivalent SRE. In particular, consider the following SRE.

$$\begin{array}{lcll} X[z] & = & z \in D^X & : \quad g_X(X[f_{XX}(z)], Y[f_{XY}(z)], \ldots) \\ Y[z] & = & z \in D^Y & : \quad g_Y(X[f_{YX}(z)], Y[f_{YY}(z)], \ldots) \end{array} \tag{4}$$

We would like to construct a semantically equivalent SRE where the domain of X is now transformed by some function $\mathcal{T}$. The required SRE is given below (a proof of a general form of this important result is given in Section 5).

$$\begin{array}{lcll} X[z] & = & z \in \mathcal{T}(D^X) & : \quad g_X(X[\mathcal{T} \circ f_{XX} \circ \mathcal{T}'(z)], Y[f_{XY} \circ \mathcal{T}'(z)], \ldots) \\ Y[z] & = & z \in D^Y & : \quad g_Y(X[\mathcal{T} \circ f_{YX}(z)], Y[f_{YY}(z)], \ldots) \end{array} \tag{5}$$

provided that $\mathcal{T}$ admits a left inverse in the context, D^X, i.e., that there exits a function $\mathcal{T}'$ such that, $\forall z \in D^X, \mathcal{T}(\mathcal{T}'(z)) = z$, i.e., $\mathcal{T} \circ \mathcal{T}' = \mathrm{Id}$. An important special (but not the only) case is when $\mathcal{T}(z) = Tz + t$ for some unimodular matrix, T and a constant vector, t. Also note that SARE's are *closed* under affine COBs, i.e., if $\mathcal{T}(z) = Tz + t$ for some constant matrix T, and vector t, the resulting SRE is also an SARE.

Definition 4. *Finally, we define the* **reduced dependence graph**, RDG *of an* SRE *as the (multi) graph with a node corresponding to each variable in the* SRE. *For each occurrence of a variable say, Y on the rhs of the equation defining say, X, there is a directed edge from X to Y. The edge is labeled with a pair, $\langle \mathcal{D}_i^X, f \rangle$ specifying the (sub) domain and the associated dependence function. The* RDG *will be an important tool for our analysis of* SRE*'s.*

4 Exact data-flow analysis of ACL's

We will now discuss how to to determine the true dependences of an ACL. Before we proceed, let us emphasize a "golden rule" that we will respect in our entire approach. During the analysis of our ACL, the resulting SARE, and implicitly, its data-flow graph, we do not *explicitly construct* this graph. This rule is motivated by a number of factors.

- The graph is usually too big, as compared to the size of the original code, to be easily manipulated by the compiler/parallelizer. For example, a matrix multiplication program has only a few lines of code but they specify about n^3 operations (and hence, a n^3-node data-flow graph). It is unreasonable to expect that compiler to explicitly construct this n^3-node graph for analysis purposes.

- More importantly, for parametric programs it is usually not (completely) known statically (i.e., at compile-time). For our matrix multiplication example, the data-flow graph for a 10×10 input matrix is distinct from that for a 100×100 matrix. The size of the matrix is a *parameter* of the program and is not known at compile-time. Any compilation/parallelization method that requires explicitly constructing the data-flow graph of the program will only be able to produce code after the size parameter is instantiated!

- Finally, even if we were to accept these limitations and construct the graph explicitly, it would not be very useful in producing efficient code. The code would need to enumerate each of the operations explicitly, and hence would correspond to a complete "unrolling" of the loops and would have an unacceptably large size.

The implication of our golden rule is that we cannot directly use the conventional and well developed methods for scheduling and mapping computations specified as *task graphs* to parallel machines. We need to exploit the regularity of our programs and work with a *reduced* representation of the data-flow graph.

We now seek to identify the true dependences of the computations of our ACL. For this, we will render explicit the results computed by each *operation* of the program, and construct an SARE which has the same semantics as the original ACL. Consider an assignment statement

$$\mathrm{S_i}: \quad \mathtt{X}[\mathtt{f}^0_\mathtt{i}(\mathtt{z},\mathtt{p})] = \mathtt{g_i}(\ldots\ \mathtt{Y}[\mathtt{f}^\mathtt{k}_\mathtt{i}(\mathtt{z},\mathtt{p})]\ldots)$$

Since each operation of the program is uniquely identified by an assignment statement and the values of the surrounding loop indices, the variables of our SARE are simply (the unique labels of) the assignment statements in the ACL. Their domains are the corresponding *iteration domains*, D_i. This determines the lhs of our SARE. Hence, what we need to determine is the rhs of the equations of our SARE. Obviously, the function g_i (the function computed by the expression on the rhs of the assignment) simply carries over to our SARE. In order to determine the arguments to g_i, we need to resolve the following question:

Problem 1: *For each read,* $\mathtt{Y}[f_i^k(z,p)]$ *of a variable (array)* Y*, with an access function* f_i^k *on the rhs of* S_i*, find the source of the value: which* ***operation*** *(i.e., which iteration of which statement) wrote the value being read?*

In general, this is a function of z, and indeed it is this function that will be the dependence function of our SARE. Our solution is some instance, say z', of an assignment statement, S_j which has the variable Y (accessed with some function f_j^0) on its lhs. We consider all such statements as candidates and address them one by one. Each of them is executed many times and possibly writing to many different memory addresses, and moreover the same memory address is (over)written many times.

For each candidate statement, our solution is one of possibly many instances z' that satisfy the following conditions (since our solution is a function of z, we also include constraints that z must satisfy):

$$\left\| \begin{array}{lcl} z & \in & D_i \\ z' & \in & D_j \\ f_i^k(z,p) & = & f_j^0(z',p) \\ \langle S_j, z' \rangle & \lhd & \langle S_i, z \rangle \end{array} \right. \tag{6}$$

The first two constraints ensure that we are dealing with valid operations, the third one ensures that the two operations in question address the same memory location, and the final one states that the operation $\langle S_j, z' \rangle$ *precedes* the operation $\langle S_i, z \rangle$ in the order of execution of the original ACL. Observe that these conditions (6) are not yet complete: they admit multiple valid points, $z' \in D_j$ that precede $\langle S_i, z \rangle$ and that write into the same memory location that is read by $\langle S_i, z \rangle$. Of these, we have to find the most recent one.

In order to do so, we first observe that the final constraint in (6) is not a simple affine (in)equality. Nevertheless, since it involves the lexicographic precedence between index points, it may be expressed as the disjunction (union) of a finite number of such constraints. Hence the set of possible solutions, z' that we need to consider is a finite union of polyhedra. Each one is of dimension $n_i + n_j + m$ (the index variables that are involved are those in the respective domains of S_i (i.e., z), of S_j (i.e., z'), and the m parameters of the ACL. We will view this as an n_j-dimensional polyhedron, but *parameterized* with the $n_i + m$ other indices. It is therefore clear that its vertices are piece-wise affine functions of z and p.

Now, let us return to the problem of determining the *most recent* point z' that satisfies the constraints (6). For each polyhedron (obtained by the decomposition of $\lhd$ into a disjunction of (in)equalities), this is nothing but the point which is the "last" one to be executed in the sequential execution order, i.e., its Lmax, which has to be one of its vertices—a piece-wise affine function of z and p. Among the different polyhedra, the most recent one is therefore the Lmax of their vertices, which is also a piece-wise affine function of z and p.

Finally, we return to the comparison of such solutions for different candidates S_j (recall that there may be more than one statements that write into the array variable `Y` in the ACL). We simply take the Lmax of each of the candidate solutions, and hence we may conclude that the overall solution to Problem 1 as posed above is piece-wise affine function of z and p. It may be automatically computed by a tool capable of manipulating parameterized polyhedra, or a parametric integer linear programming solver.

Although the overall idea is fairly simple, the details are intricate and are best left to an automatic program analysis tool. We illustrate the analysis method by means of the example below.

Problem: In the program of Figure 1, determine the source of `s` on the rhs of `S3`.

Solution: We have 4 statements, of which `S1` is special (it's domain is $\mathcal{D}_0 = \mathcal{Z}^0$ the empty polyhedron). The other relevant domains are $\mathcal{D}_2 = \mathcal{D}_4 = \{i \mid 1 \leq i \leq n-1\}$, and $\mathcal{D}_3 = \{i, j \mid 1 \leq i < j \leq n\}$. The following source analyses are needed: `s` and `x` in `S3`, and `s` in `S4`. The main point to note is that the i loop *goes down* from $n-1$ to 1, and hence we have to be careful about our precedence order, $\prec$, and that we will not always look for the lexicographic *maximum*, Lmax (in the i dimension it will be minimum.

Since two statements write into `s`, $\mathrm{Src}(\mathtt{s}, \mathtt{S3}) = \mathrm{Last}(\langle \mathtt{S2}, f_1(i,j)\rangle, \langle \mathtt{S3}, f_2(i,j)\rangle)$, where $f_1(i,j)$ and $f_2(i,j)$ are respectively[2]

$$f_1(i,j) \quad = \quad \mathrm{Last}\left\{ (i' \mid i,j) \left\| \begin{array}{l} i' \in \mathcal{D}_2 \\ (i,j) \in \mathcal{D}_3 \\ \langle \mathtt{S2}, i'\rangle \prec \langle \mathtt{S3}, (i,j)\rangle \end{array} \right. \right\} \tag{7}$$

$$f_2(i,j) \quad = \quad \mathrm{Last}\left\{ (i', j' \mid i,j) \left\| \begin{array}{l} (i', j') \in \mathcal{D}_3 \\ (i,j) \in \mathcal{D}_3 \\ \langle \mathtt{S3}, (i', j')\rangle \prec \langle \mathtt{S3}, (i,j)\rangle \end{array} \right. \right\} \tag{8}$$

Using the definition of the $\prec$ relation[3], and our knowledge of the program text, we see that $\langle \mathtt{S2}, i'\rangle \prec \langle \mathtt{S3}, (i,j)\rangle$ if and only if $i' \geq i$. Hence

$$f_1(i,j) \quad = \quad \mathrm{Last}\left\{ (i' \mid i,j) \left\| \begin{array}{l} i' \in \mathcal{D}_2 \\ (i,j) \in \mathcal{D}_3 \\ i \leq i' \end{array} \right. \right\} \tag{9}$$

$$\quad = \quad \mathrm{Last}\left\{ (i' \mid i,j) \parallel i \leq i' \leq n; 1 \leq i < j \leq n \right\} \tag{10}$$

[2]For notational simplicity, vectors are written as rows, and the bar separates the parameters, i.e., we seek the last i' as a function of i and j such that the stated constraints are satisfied.

[3]Observe the sense of the inequality.

```
S1:    x[n] := b[n]/u[n,n];
       for i = n-1 down to 1 do
S2:        s := 0;
           for j = i+1 to n do
S3:            s := s + x[j] * u[i, n-j];
           enddo
S4:        x[i] := (b[i] - s) / u[i,i]
       enddo
```

Figure 1: An affine control loop to solve an upper triangular system of equations.

This set of points is viewed as a family 1-D polyhedra (line segments indexed by i') *parameterized* by points in a 2-D parameter polyhedron, $\mathcal{D}_3$. At all points in $\mathcal{D}_3$, the line segment is from i to n, and here the Last operation is equivalent to finding the lexicographic *minimum*. Hence $f_1(i,j) = i$.

Now, to find $f_2(i,j)$, we observe that $\langle \texttt{S3}, (i', j') \rangle \prec \langle \texttt{S3}, (i, j) \rangle$ is equivalent to $i' > i \vee (i' = i \wedge j' < j)$, a disjunction of two sets of inequality constraints, and hence

$$
\begin{aligned}
f_2(i,j) &= \text{Last}\left(\left\{(i',j' \mid i,j) \left\| \begin{array}{l}(i',j') \in \mathcal{D}_3\\ (i,j) \in \mathcal{D}_3\\ i < i'\end{array}\right.\right\}\right. \\
&\qquad\qquad \left.\bigcup \left\{(i',j' \mid i,j) \left\| \begin{array}{l}(i',j') \in \mathcal{D}_3\\ (i,j) \in \mathcal{D}_3\\ i' = i; j' < j\end{array}\right.\right\}\right) \qquad (11)\\
&= \text{Last}\left(\text{Last}\left\{(i',j' \mid i,j) \left\| \begin{array}{l}(i',j') \in \mathcal{D}_3\\ (i,j) \in \mathcal{D}_3\\ i < i'\end{array}\right.\right\},\right. \\
&\qquad\qquad \left.\text{Last}\left\{(i',j' \mid i,j) \left\| \begin{array}{l}(i',j') \in \mathcal{D}_3\\ (i,j) \in \mathcal{D}_3\\ i' = i; j' < j\end{array}\right.\right\}\right) \qquad (12)\\
&= \text{Last}\big(\text{Last}\{(i',j' \mid i,j) \,\|1 \le i < i' < j' \le n; i < j \le n\} \\
&\qquad\qquad \text{Last}\{(i',j', \mid i,j) \,\|1 \le i' = i < j \le n; j' < j\}\big) \qquad (13)
\end{aligned}
$$

This is $\text{Last}(f_2'(i,j), f_2''(i,j))$ where

$$
f_2'(i,j) = \begin{cases} \{i,j \mid i = n-1; j = n\} & : \perp \\ \{i,j \mid 1 \le i < j \le n; i \le n-2\} & : (i+1,n) \end{cases} \qquad (14)
$$

$$
f_2''(i,j) = \begin{cases} \{i,j \mid 1 \le i = j-1 \le n-1\} & : \perp \\ \{i,j \mid 1 \le i < j-1 \le n-1\} & : (i,j-1) \end{cases} \qquad (15)
$$

$$
\qquad (16)
$$

and hence

$$f_2(i,j) = \begin{cases} \{i,j \mid i = n-1; j = n\} & : \ \mathrm{Last}(\perp, \perp) \\ \{i,j \mid 1 \le i = j-1 \le n-2\} & : \ \mathrm{Last}((i+1,n), \perp) \\ \{i,j \mid 1 \le i < j-1 \le n-1; i \le n-2\} & : \ \mathrm{Last}((i+1,n),(i,j-1)) \end{cases} \tag{17}$$

$$= \begin{cases} \{i,j \mid i = n-1; j = n\} & : \ \perp \\ \{i,j \mid 1 \le i = j-1 \le n-2\} & : \ (i+1,n) \\ \{i,j \mid 1 \le i < j-1 \le n-1; i \le n-2\} & : \ (i,j-1) \end{cases} \tag{18}$$

Finally, remember that

$$\begin{aligned} \mathrm{Src}(\mathtt{s}, \mathtt{S3}) &= \mathrm{Last}\left(\langle \mathtt{S2}, f_1(i,j)\rangle, \langle \mathtt{S3}, f_2(i,j)\rangle\right) \\ &= \mathrm{Last}\left(\langle \mathtt{S2}, i\rangle, \left\langle \mathtt{S3}, \begin{cases} \{i,j \mid i = n-1; j = n\} & : \ \perp \\ \{i,j \mid 1 \le i = j-1 \le n-2\} & : \ (i+1,n) \\ \{i,j \mid 1 \le i < j-1 \le n-1; i \le n-2\} & : \ (i,j-1) \end{cases} \right\rangle \right) \\ &= \mathrm{Last}\left(\langle \mathtt{S2}, i\rangle, \begin{cases} \{i,j \mid i = n-1; j = n\} & : \ \langle \mathtt{S3}, \perp\rangle \\ \{i,j \mid 1 \le i = j-1 \le n-2\} & : \ \langle \mathtt{S3}, (i+1,n)\rangle \\ \{i,j \mid 1 \le i < j-1 \le n-1; i \le n-2\} & : \ \langle \mathtt{S3}, (i,j-1)\rangle \end{cases} \right) \\ &= \begin{cases} \{i,j \mid i = n-1; j = n\} & : \ \mathrm{Last}(\langle \mathtt{S2}, i\rangle, \langle \mathtt{S3}, \perp\rangle) \\ \{i,j \mid 1 \le i = j-1 \le n-2\} & : \ \mathrm{Last}(\langle \mathtt{S2}, i\rangle, \langle \mathtt{S3}, (i+1,n)\rangle) \\ \{i,j \mid 1 \le i < j-1 \le n-1; i \le n-2\} & : \ \mathrm{Last}(\langle \mathtt{S2}, i\rangle, \langle \mathtt{S3}, (i,j-1)\rangle) \end{cases} \\ &= \begin{cases} \{i,j \mid i = n-1; j = n\} & : \ \langle \mathtt{S2}, i\rangle \\ \{i,j \mid 1 \le i = j-1 \le n-2\} & : \ \langle \mathtt{S2}, i\rangle \\ \{i,j \mid 1 \le i < j-1 \le n-1; i \le n-2\} & : \ \langle \mathtt{S3}, (i,j-1)\rangle \end{cases} \\ &= \begin{cases} \{i,j \mid 1 \le i = j-1 \le n-1\} & : \ \langle \mathtt{S2}, i\rangle \\ \{i,j \mid 1 \le i < j-1 \le n-1; i \le n-2\} & : \ \langle \mathtt{S3}, (i,j-1)\rangle \end{cases} \end{aligned} \tag{19}$$

We now have determined the precise source of `s` on the right hand side of statement `S3` as a function of its loop iteration indices (and the system parameters). Essentially, it states that for the very first iteration (i.e., for $j = i+1$) the producer is the i-th iteration of statement `S2`, otherwise it is the "previous", i,e, the $(i, j-1)$-th iteration of `S3` itself, as can be easily verified *end of example*

5 SARE's: merely an intermediate form?

In this section, we argue that the formalism of SARE's, augmented with reduction operations, is more than just an intermediate form for the parallelization of ACL's, but is interesting in its own right. We present an equational language, Alpha based on this formalism, describe its expressive power, and briefly explain its *denotational semantics.* We also describe some of the analyses that are enabled by this formalism (some that would be extremely difficult for a imperative language).

Alpha, designed by Mauras in 1989 in the context of systolic array synthesis at Irisa (France), is a strongly typed data-parallel functional language based on SARE's. MMAlpha is a prototype transformation system (also developed at Irisa, and available under the Gnu Public License at `http://www.irisa.fr/cosi/ALPHA`) for reading and manipulating Alpha programs through correctness-preserving transformations and eventually generating VHDL description of regular, systolic VLSI circuits or (sequential or parallel) code for programmable (i.e., instruction-set) processors.

Alpha variables are type declared at the beginning of a program, and represent "polyhedral shaped" multidimensional arrays. The polyhedra specified in the declaration are called the *domains* of the variables. For example, a (strictly) lower triangular, real matrix is specified by the following declaration:[4]

```
A: {i,j| 0<j<i<=N} of real
```

To introduce the main features of Alpha, consider the problem of solving, using forward substitution, a system of linear inequalities, $Ax = b$, where A is a lower triangular $n \times n$

[4]This syntax is to be read as "the set of i, j such that" the specified linear inequalities are satisfied. In specifying the inequalities, we may group expressions using parentheses. For example, `(i+j,j+10)<=N` specifies a domain where both `i+j` and `j+10` are no greater than `N`.

matrix with unit diagonal. A high-level, mathematical description of the program would be

$$\text{for } i = 1 \ldots n, \quad x_i = \begin{cases} \text{if } i = 1 & b_j \\ \text{if } i > 1 & b_i - \sum_{j=1}^{i-1} A_{i,j} x_j . \end{cases} \tag{20}$$

The corresponding ALPHA program (Figure 2) is identical, except for syntactic sugar. The first line names the system and declares that it has a positive integer parameter, `N`, which can take any integer value greater than 1. The next three lines are declarations of the input and output variables of the system (respectively, before and after the `returns` keyword). The domain of `A` is triangular, while `B` and `X` are one-dimensional variables (vectors). A system may also have local variables (not present here), which are declared after the system header, using the keyword `var`. The body of the program is a number of equations delineated by the `let` and `tel` keywords. The rhs of an equation is an expression, following the syntax given in Table 1.

In our example, we have a single equation, almost identical to (20) above. The `case` construct has the usual meaning and allows us to define conditional expressions. The `restrict` construct has the syntax `<domain> : <expr>`, and denotes the expression `<expr>` but restricted to the subset of index points in `<domain>`. The `reduce` construct corresponds to the summation in (20) and has three parts: an associative and commutative operator, a projection function, and an expression. Here the operator is `+`. The projection is `(i,j->i)`, and denotes (intuitively) the fact that within the body of the `reduce`, there are two indices, `i` and `j`, but only `i` is visible outside the scope of the `reduce`, i.e., the two-dimensional expression of the body is projected by the mapping `(i,j->i)` to a one-dimensional one. The body of our `reduce` construct is the expression, `A * X.(i,j->j)`.

Here, `(i,j -> j)`, is a *dependency function* and denotes the fact that to compute the body at `[i,j]`, we need the value of `X` at index point `[j]` (the dependency on `A` is not explicitly written—it is the identity). Dependencies are important in ALPHA. They have the syntax `(Idx, ...  -> IdxExpr, ...)`, where each `Idx` is an index name, and `IdxExpr` is an *affine* expression of the system parameters and the `idx`'s. ALPHA and MMALPHA use this syntax for specifying a multidimensional affine function in many different contexts (e.g., the projection function in a `reduce` expression). This syntax can be viewed as a special kind

```
system ForwardSubstitution : { N | N>1 }         -- comments are like this
               ( A : { i,j | 0<j<i<=N } of real; -- a 2D input variable
                 B : { i   | 0<i<=N } of real)   -- a 1D input variable
       returns ( X : { i   | 0<i<=N } of real ); -- a 1D output variable
let
  X = case
        {i | i=1} : B;
        {i | i>1} : B - reduce(+, (i,j -> i), A * X.(i,j->j));
      esac;
tel;
```

Figure 2: Alpha program for the forward substitution algorithm.

⟨Exp⟩	:=	⟨Const⟩ | ⟨Var⟩	atomic expressions
		| $\oplus$ ⟨Exp⟩ | ⟨Exp⟩ $\oplus$ ⟨Exp⟩	unary/binary pointwise ops
		| `if` ⟨Exp⟩ `then` ⟨Exp⟩ `else` ⟨Exp⟩	a ternary pointwise op
		| ⟨Dom⟩ : ⟨Exp⟩	restrictions
		| `case` ⟨Exp⟩;...⟨Exp⟩; `esac`	case expressions
		| ⟨Exp⟩.⟨Dep⟩	dependence expressions
		| `reduce` ($\oplus$, ⟨Dep⟩, ⟨Exp⟩)	reductions
⟨Dom⟩	:=	`{` ⟨Idx⟩ ... `|` ⟨IdxExpr⟩ `>=0 }`	domains
⟨Dep⟩	:=	`(` ⟨Idx⟩ ... `->` ⟨IdxExpr⟩ `...)`	dependences
⟨IdxExpr⟩	:=	any affine expression of indices and system parameters	

Table 1: Syntax of Alpha expressions (summary)

of lambda expression, restricted to affine mappings from $\mathcal{Z}^n$ to $\mathcal{Z}^m$. Such a function, f, may be equivalently represented by an $m \times n$ matrix A, and an m-vector a, i.e., $f(z) = Az + a$, and we will later use this form for analysis purposes.

5.1 Semantics of Alpha expressions

Since Alpha is a data-parallel language, expressions denote collections of values. Indeed *all* expressions (not just variables) can be viewed as multidimensional arrays and denote a function from indices to values. Alpha semantics consist of two parts: a semantic function and a domain. The semantic function is defined using classic methods and is fairly obvious (the only subtle point being dependencies and reductions, as described later), but the domains are a unique aspect of Alpha. *Every* (sub) expression in a program has a domain, and it can be determined from the domains of its subexpressions, as summarized below.

- *Identifiers and Constants:* An identifier simply denotes the variable it identifies, and is defined over its *declared* domain. In general this is a finite union of polyhedra. A constant expression denotes the constant itself, and its domain is $\mathcal{Z}^0$, the zero-dimensional polyhedron.

- *Pointwise Operators:* The expression $\mathtt{E} \oplus \mathtt{F}$ denotes the pointwise application of the operator $\oplus$ to the corresponding elements of the expressions `E` and `F`, and hence its domain is $\text{Dom}(\mathtt{E}) \cap \text{Dom}(\mathtt{F})$, the intersection of the domains of its subexpressions. The semantics of unary or ternary (the `if-then-else`) pointwise operators, are similar.

- *Restriction:* The expression $\mathcal{D} : \mathtt{E}$ denotes the expression `E`, but restricted to the domain $\mathcal{D}$, and hence its domain is $\mathcal{D} \cap \text{Dom}(\mathtt{E})$.

- *Case:* The expression `case E;F esac` denotes an expression which has alternative definitions (there may be more than two alternatives). Its domain is $\text{Dom}(\mathtt{E}) \cup \text{Dom}(\mathtt{F})$, the (disjoint) union of the domains of its subexpressions.

- *Dependency:* First of all, we note that the dependency $(\mathtt{z} \rightarrow f(\mathtt{z}))$ by itself, simply denotes the affine function, f. We will therefore abuse the notation somewhat to use the same nomenclature for both, the syntactic construct as well as for the corresponding semantic function.

  The expression `E.f` denotes an expression whose value at `z` is the value of `E` at $f(\mathtt{z})$. Its domain must therefore be the set of points which are mapped by f to some point in the domain of `E`. This is nothing but the preimage, of $\text{Dom}(\mathtt{E})$ by f, i.e.,

  $$\text{Dom}(\mathtt{E.f}) = \text{Pre}(\text{Dom}(\mathtt{E}), f) = f^{-1}\text{Dom}(\mathtt{E})$$

  where f^{-1} is the relational inverse of f.

- *Reductions:* The expression `reduce(`$\oplus$`, f, E)` denotes an expression whose domain is the *image* of the domain of `E` by f. Its value at any point, z, in this domain is obtained by taking all points in the domain of `E` which are mapped to z by f, and applying the associative and commutative operator $\oplus$ to the values of `E` at these points.

Note that the above semantics imply that the domain of any Alpha expression may be determined recursively in a top-down traversal of its syntax tree. Because the *declared* domains in Alpha (i.e., the domains of the leaves of the tree) are finite unions of polyhedra, and thanks to the mathematical closure properties of polyhedra and affine functions, the domain of any Alpha expression can be easily determined by using a library for manipulating polyhedra.

5.2 Alpha transformations

The denotational semantics given above describe what an Alpha expression denotes or means, without necessarily showing how to compile or otherwise execute an Alpha program. For this one needs an *operational* semantics. We defer an brief review of an operational semantics to the end of this section. For now, note that the denotational semantics enable us to formally reason about Alpha programs, and to develop and prove the validity of program transformations. These program transformations are available in the MMAlpha system, and we shall describe two of the important ones here.

5.2.1 Normalization

A number of properties can be proved based on the denotational semantics of Alpha expressions. For example, we can show that for any expression `E`, and dependences f_1 and f_2, the expression $\mathtt{E.f_1.f_2}$ is semantically equivalent to `E.f`, where $f = f_1 \circ f_2$ is the composition[5] of f_1 and f_2.

Although a large number of such "transformation rules" could be developed and proposed to the user, it turns out that a certain set of rules is particularly useful to "simplify" any Alpha expression into what is called a *normal form*, or the `case-restriction-dependency` form. It consists of an (optional) outer case, each of whose branches is a (possibly restricted) simple expression. A simple expression consists of (possibly a reduction of) either variables or constants composed with a single dependency function (which may be omitted, if it is the identity), or pointwise operators applied to such subexpressions. This simplification is obtained by a set of rewrite rules which combine dependencies together, eliminate empty do-

[5]Note that function composition is right-associative, i.e., $f_1 \circ f_2(z) = f_1(f_2(z))$.

main expressions, introduce new local variables to define variables for certain subexpressions (to remove nested reductions) etc.

Let `X = {i,j | 0<i,j<=N} : X.(p,q->q,p) + 0.(i,j->)` be an equation in an Alpha program (observe that the domain of the constant `0` is $\mathcal{Z}^0$, whose preimage by the function $(i, j \rightarrow)$ is $\mathcal{Z}^2$ which is coherent with the rest of the expression). Since the right-hand side of this equation is normalized, we can rewrite the equation as follows:

- rename the indices in the restrictions and in the dependencies to be identical (eg. by by replacing `p` and `q` by `i` and `j`, respectively)
- move the index names in the dependencies (left of the "`->`") and in domains (left of the "`|`") to the left of the entire equation

This yields the following "sugared" syntax, called the *array notation*, which is often more readable (indeed, all subsequent examples in this chapter are presented in this notation).

```
X[i,j] = {| 0<i,j<=N} : X[i,j] + 0[]
```

Also observe that sare's as defined in Section 3 constitute a proper subset of Alpha programs, namely those that do not contain reductions, and those that are normalized.

5.2.2 Generalized Change-of-Basis

Perhaps the most important transformation in the Alpha system is the **change of basis** (CoB). The intuition behind it is as follows. Since an Alpha variable can be viewed as a multidimensional array defined over a polyhedral domain, we should be able to "move" (or otherwise change the "shape" of) its domain and construct an equivalent program. We now develop such a transformation.

Let $\mathcal{T}$ and $\mathcal{T}'$ be functions such that for all points z in some set S of index points, $\mathcal{T}' \circ \mathcal{T}(z) = z$. Note that $\mathcal{T}$ and $\mathcal{T}'$ may not even be affine, but even in the case when they are, we do not insist that they be square, nor that $\mathcal{T}' \circ \mathcal{T}$ be the identity. We say that $\mathcal{T}'$ is the *left* inverse of $\mathcal{T}$ in the *context* of S. The following can easily be proved from the semantics of Alpha expressions as defined above.

Remark 1. *For any* Alpha *expression* $\mathtt{E}$*, let* $\mathcal{T}$ *and* $\mathcal{T}'$ *be such that* $\mathcal{T}'$ *is the left inverse of* $\mathcal{T}$ *in the context of* $\mathrm{Dom}(\mathtt{E})$*. Then* $\mathtt{E}$ *is semantically equivalent to* $\mathtt{E}.\mathcal{T}'.\mathcal{T}$*, and any occurrence of the former anywhere in the program may be replaced by the latter.*

This implies in particular, that if we choose $\mathtt{E}$ as the subexpression consisting of just the variable $\mathtt{X}$, then *every* occurrence of $\mathtt{X}$ can be replaced by $\mathtt{X}.\mathcal{T}'.\mathcal{T}$, without affecting the program semantics. Moreover, if $\mathtt{Expr}$ is the entire rhs of the equation defining $\mathtt{X}$, then

$$\mathtt{X}.\mathcal{T}' = \mathtt{Expr}.\mathcal{T}' = \mathtt{X}' \text{ (say)}$$

We may therefore introduce a new local variable $\mathtt{X}'$ and define its domain to be $\mathrm{Pre}(D_X, \mathcal{T}')$, and the rhs of its defining equation to be $\mathtt{Expr}.\mathcal{T}'$. Next, we replace every occurrence of the subexpression $\mathtt{X}.\mathcal{T}'$ in the program by $\mathtt{X}'$ (since $\mathtt{X}.\mathcal{T}' = \mathtt{X}'$) and finally, since $\mathtt{X}$ is no longer used in the program, drop it, and then rename the $\mathtt{X}'$ to be $\mathtt{X}$. This argument is embodied in the following theorem.

Theorem 1. *In an* Alpha *program with a local variable* $\mathtt{X}$ *declared over a domain* $\mathtt{D}$*, let* $\mathcal{T}$ *and* $\mathcal{T}'$ *be such that* $\mathcal{T}'$ *is the left inverse of some* $\mathcal{T}$ *in the context of* $\mathtt{D}$*. The following transformations yield a semantically equivalent program.*

- *Replace the domain of declaration of* $\mathtt{X}$ *by* $\mathrm{Pre}(\mathtt{D}, \mathcal{T}')$*.*
- *Replace all occurrences of* $\mathtt{X}$ *on the right-hand side of* any *equation by* $\mathtt{X}.\mathcal{T}$*.*
- *Compose the entire rhs expression of the equation for* $\mathtt{X}$ *with* $\mathcal{T}'$

Many program transformations such as alignment, scheduling, processor allocation, etc., can be implemented as an appropriate CoB. Moreover, we can see that the rules for CoB for SRE's in Section 3 are merely a special case where the resulting program is normalized after applying the above rules.

5.3 Reasoning about Alpha programs

The functional/equational nature of Alpha provides us important advantages not available in a conventional imperative language. These include the ability to formally reason about programs, the possibility to systematically derive programs from mathematical specifications, to use advanced program analysis techniques such as abstract interpretation, etc. We illustrate one of them here by showing how formal equivalence properties of Alpha programs can be proved systematically. Consider the following two recurrences, both defined over the same domain, $D = \{i, j, k \mid 1 \leq i, j \leq n; 0 \leq k \leq n\}$.

$$T[i,j,k] \quad = \quad \begin{cases} \{\mid k=0\} & : \ 0 \\ \{\mid i=j=k\} & : \ 1+T[i,j,k-1] \\ \{\mid i=k\neq j\} & : \ 1+\max(T[k,k,k],T[i,j,k-1]) \\ \{\mid j=k\neq i\} & : \ 1+\max(T[i,j,k-1],T[k,k,k]) \\ \{\mid i,j\neq k; k>0\} & : \ 1+\max(T[i,j,k-1], \\ & \qquad\qquad T[i,k,k],T[k,j,k-1]) \end{cases} \tag{21}$$

$$T'[i,j,k] \quad = \quad \begin{cases} \{\mid k=0\} & : \ 0 \\ \{\mid i=j=k\} & : \ 3k-2 \\ \{\mid i=k\neq j\} & : \ 3k-1 \\ \{\mid j=k\neq i\} & : \ 3k-1 \\ \{\mid i,jk; k>0\} & : \ 3k \end{cases} \tag{22}$$

The definition of T is recursive, while that of T' is in “closed form”. However, it is probably not immediately obvious that the two functions compute the same result. We would like to formally prove this. Specifically, we would like to show that for any point z in D, $T[z] = T'[z]$. This can be done manually by an inductive argument (essential a structural induction on the recursive structure of T).

In order to do this mechanically with a theorem prover and MMAlpha, we first write an Alpha program which has three variables, all defined over the same domain, D. The first two are integer-typed variables, T and T' as defined above. The third is a Boolean variable, `Th`, the theorem that we seek to prove, i.e., the rhs of its equation is simply the expression `T = T'`. We would like to show that `Th` has the value “True” everywhere in its domain. Our proof proceeds as follows.

- We first substitute the definitions of `T` and `T'` in the rhs of `Th` (this is a provably correct transformation, since Alpha is functional, and is trivially simple to implement in MMAlpha), yielding the following equation for `Th` (a normalization has been done to render the program more readable).

$$\mathtt{Th}[i,j,k] \;=\; \begin{cases} \{\mid k=0\} & : \; 0=0 \\ \{\mid i=j=k\} & : \; 3k-2 = 1+T[i,j,k-1] \\ \{\mid i=k\neq j\} & : \; 3k-1 = 1+\max(T[k,k,k],T[i,j,k-1]) \\ \{\mid j=k\neq i\} & : \; 3k-1 = 1+\max(T[i,j,k-1],T[k,k,k]) \\ \{\mid i,j;k>0\} & : \; 3k = 1+\max(T[i,j,k-1],T[i,k,k],T[k,j,k-1]) \end{cases} \qquad (23)$$

- Next we make the inductive hypothesis namely that the theorem is true for the recursive calls in the definition of `T`, so that `T` may be replaced by `T′` on the rhs of (23). In general, this is *not* a correctness-preserving transformation in MMAlpha. It is simply used here in the context of proving a certain property, and corresponds to *making* the induction hypothesis.

- We next substitute the newly introduced instances of `T′` by the closed form definition from Eqn. (22) and simply normalize the program. We obtain an equation (not shown here) with a number of case branches, each of which is an equality of some arithmetic expressions. Using a standard theorem prover with some knowledge about arithmetic operations, it is fairly easy to show that these are all tautologies.

Actually, for our example, we can even do a little better, since the closed form of the expressions are all *affine* functions of the indices. For such programs, the *entire* proof can be completely performed in MMAlpha, and indeed, if we simply normalize the program after making the inductive hypothesis, we simply obtain the following equation for `Th`.

$$\mathtt{Th}[i,j,k] = \mathtt{True}, \qquad (24)$$

which is exactly what we wanted to prove (and why the we didn't show it earlier).

6 Scheduling in the polyhedral model

We now describe how to resolve one of the fundamental analysis questions, namely assigning an execution date to each instance of each variable in the original SARE. Remember that our golden rule implies that we work on the *compact representation* of the program, rather than the computation graph that it induces (called the *extended* dependence graph, EDG). This means that the schedule cannot be specified by enumerating the time instances at which each operation is executed, but rather as a closed form function. We will first give a classic technique (the wavefront method) to determine schedules for a *single* URE. Then, we will describe how this can be extended to deal with SURE's, present some of the limitationss of these extensions, and develop the algorithms used to determine more general schedules for SURE's. Next we show how these scheduling algorithms can be carried over to ARE's and SARE's, by exploiting the fact that the domains of the variables are polyhedra.

Since our schedule is to be expressed in closed form, the time instant at which an operation $O = \langle S_i, z\rangle$ (or equivalently, a variable X at point z) is executed is given by the function $\tau(O)$ or $\tau_X(z)$. In the polyhedral model we restrict ourselves to *affine schedules*, defined as follows (recall that p is the l-dimensional space of size parameters):

$$\tau(\langle S_X, z\rangle) = \Lambda_X z + \alpha_X + \Lambda'_X p \tag{25}$$

where Λ_X (resp. Λ'_X) is a constant $k \times n_X$ (resp. $k \times l$) integer matrix, and α_X is an integral k-vector. Here, k is called the *dimension* of the schedule, and n_X is the number of dimensions in the iteration domain of S_X). Such a schedule maps every operation of the ACL to a k-dimensional integer vector. Since there exists a natural total order relation—the lexicographic order—over such vectors we can interpret these vectors as a "time instant".

There are a number of special cases that we shall consider. If $k = 1$ we have what are called 1-dimensional schedules—these are the simplest to understand and visualize, and we shall initially focus on this class. Another common case is when the schedule function is the *same* for all the variables of the SARE (regardless of the dimension of the schedule, but usually for 1-dimensional schedules). We call them *variable-independent* schedules[6], and drop the subscript i in Eqn. (25). Note that variable-independent schedules can only be defined for SRE's where all variables have the *same* number of dimensions. A slightly more general case is when the *linear* part of the affine function is the same for all variables, but the constant α_X may be different. Such schedules are called *shifted linear schedules.*

[6]Sometimes we use the term *variable-dependent* schedules when we are *not* restricting ourselves to variable-independent schedules.

1-dimensional (variable-independent) schedules have a nice, intuitive geometric interpretation. The set of points computed at a given instant t, are precisely those that satisfy $\Lambda z + \Lambda' p + \alpha = t$. Since, for a given problem instance, $\Lambda' p$ is fixed, these points are characterized by the equation $\Lambda z =$ constant, which defines a family of hyperplanes whose normal vector is Λ. Such schedules are therefore visualized as "wavefronts" or "iso-temporal" hyperplanes through the iteration space. We defer the geometric visualization of multidimensional schedules (i.e., schedules with $k > 1$) to later.

Since the schedule maps operations to k-dimensional integer vectors, it is obvious that any schedule is valid if and only if the total order induced by the schedule respects the causality constraints of the computation, as described below.

Remark 2. *A schedule as defined in Eqn. (25) is valid if and only if for every edge $\langle D, f \rangle$ from node X to Y in the* RDG*,*

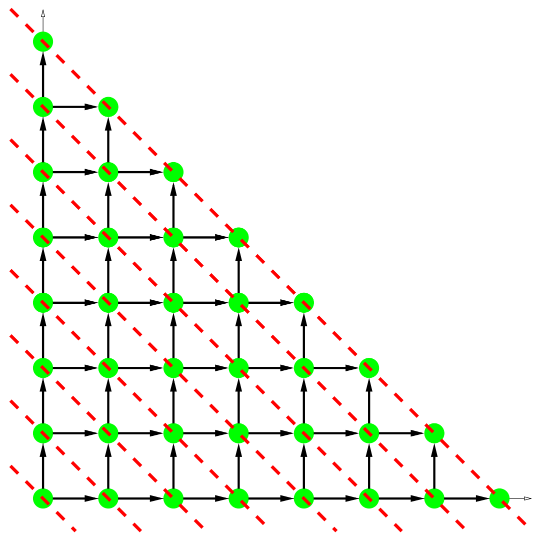
$$\begin{aligned} &\forall z \in D, \\ &\quad \Lambda_X z + \alpha_X + \Lambda'_X p \ \succ\ \Lambda_Y f(z) + \alpha_Y + \Lambda'_Y p \end{aligned} \tag{26}$$

In formulating this constraint we assume a machine architecture that can execute any instance of the rhs of any statement in the original ACL in one "time step". The main goal of the scheduling algorithms is to express the potentially unbounded instances of the above constraints (26) in a compact manner by exploiting properties of the polyhedral model.

6.1 Scheduling a single URE: 1-dimensional schedules

Consider a single URE with appropriate "boundary conditions" (not shown):

$$\forall z \in D \quad X[z] = g(X[z + d_1] \ldots X[z + d_s]) \tag{27}$$



$$\begin{aligned} X[i,j] &= g(X[i-1,j], X[i,j-1] \\ t(i,j) &\equiv ai+bj+\alpha \end{aligned}$$

Schedule validity conditions

$$\begin{aligned} [a,b][0,-1]^T &< 0 \\ [a,b][-1,0]^T &< 0 \\ \alpha &\geq 0 \end{aligned}$$

i.e., $\{\mathtt{a},\mathtt{b},\alpha \mid \mathtt{a},\mathtt{b} > \mathtt{0}, \alpha \geq \mathtt{0}\}$

The constraint on α is because $t(i,j)$ must be positive at all points in the domain. It is easy to see that the *optimal*, i.e., the fastest schedule is $t(i,j) = i+j$.

Figure 3: Scheduling a single URE with a 1-dimensional schedule.

The scheduling constraints of Eqn. (26) reduce to: for $j = 1 \ldots s$

$$\begin{aligned} &\forall z \in D, \\ &\qquad \Lambda z + \alpha + \Lambda' p \;>\; \Lambda(z+d_j) + \alpha + \Lambda' p \\ &\qquad \text{i.e., } \forall z \in D, \\ &\qquad\qquad \Lambda d_j \;<\; 0 \qquad (28) \end{aligned}$$

We observe that the z “cancels out” from the constraints, thus giving us a finite number of constraints. The feasible space of valid schedules is thus a polyhedron (indeed, a *cone*). There are (potentially unboudedly) many valid schedules, and by introducing an appropriate linear cost function (eg. the total execution time) we can easily formulate scheduling as an integer linear programming problem, and draw from well established techniques to find optimal schedules. An example of this technique is given in Figure 3.

6.2 Scheduling SURE’s: variable-dependent schedules

With SURE’s, we may use the same technique as above and seek a single, variable-independent schedule. The formulation of the schedule constraints then remains identical to that in

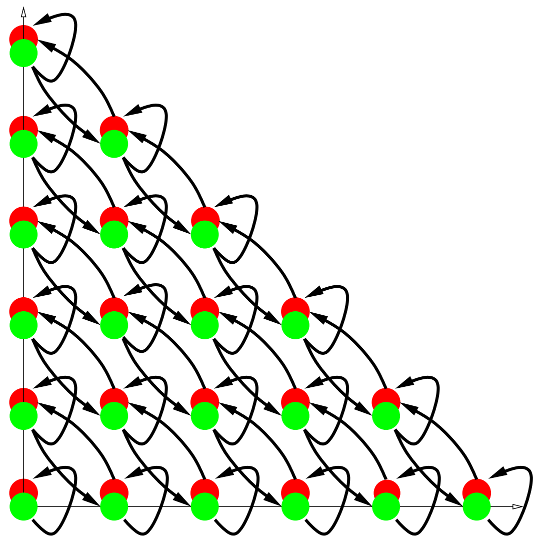
Eqn. (28) above. However, variable-independent schedules are restrictive since some SURE's (eg. the SURE of Eqns. (29-32) below) may not admit such a schedule. Here, the computation at a point $[i, j]$ needs another result at the same point, and hence there is a dependence vector $\vec{0}$. It does not matter that the two variables involved are distinct, as far as the scheduling constraints are concerned.

$$
\begin{aligned}
y_i &= Y[i, n-1] && (29)\\
Y[i,j] &= \begin{cases} j = 0 : W[i,j] * X[i,j] \\ j > 0 : Y[i,j-1] + W[i,j] * X[i,j] \end{cases} && (30)\\
X[i,j] &= \begin{cases} j = 0 : x_i \\ j > 0 : X[i-1,j-1] \end{cases} && (31)\\
W[i,j] &= \begin{cases} i = 0 : w_j \\ i > 0 : W[i-1,j] \end{cases} && (32)
\end{aligned}
$$

A simple way out of this situation is to use *shifted linear* schedules, for which the constraints now include the α values of each variable. For the SUREof Eqns. (29-31), the optimal schedule is obtained to be $t_X(i,j) = t_W(i,j) = i + j$ and $t_Y(i,j) = i + j + 1$. However, although shifted linear schedules resolve the problem for the above SURE, they are still not general enough, as illustrated in Figure 4. The problem of determining a variable-dependent 1-dimensional schedule for an SURE can also be formulated as a linear programming problem, although the arguments are a little more intricate than those leading to Eqn. (28). Essentially, we define linear constraints that ensure that each data value "comes from a strictly preceding" hyperplane. Note that the simple geometric interpretation of a *single* family of iso-temporal wavefronts sweeping out the iteration domain seems to break down with variable-dependent schedules. This will (slightly) complicate the final code generation step, as we shall see later.

6.3 Scheduling SURE's: multidimensional schedules

Although variable-dependent, 1-dimensional affine schedules work well on SURE's like the example above, they are still too restrictive. It is not always possible to find such a schedule



$$
\begin{aligned}
X[i,j] &= f(X[i-1,j+1], Y[i,j-1]) \\
Y[i,j] &= g(Y[i+1,j-1], X[i,j])
\end{aligned}
$$

For this SURE, it can be verified from the EDG that (i) the longest path reaching (either of the two) nodes at index point $[i,j]$ must pass through *all* the points in the triangular region "below" the $i+j$ "diagonal". Since the number of such points is a quadratic function of the indices, there can be no linear schedule—there will always be a point (far enough from the origin) such that a path reaching it will exceed any proposed linear schedule.

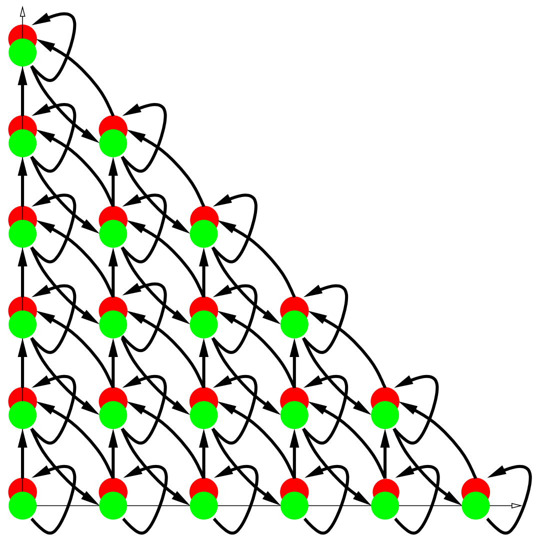
Figure 4: Limitations of variable-independent and shifted linear schedules.

for many SURE's (see Figure 5). It is therefore necessary to use the full generality of multidimensional schedules as defined earlier. There is still a nice geometric interpretation (easiest to visualize with variable-independent schedules). We interpret *each row* of the Λ matrix as defining a family of hyperplanes or wavefronts. The first row defines, say the hours, the next one the minutes, and so on. However, caveat emptor: the hours/minutes analogy is at best approximate. In lexicographic order, $\begin{pmatrix} 0 \\ x \end{pmatrix}$ can *never* precede $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$, however large we make x; but 61 minutes exceeds 1 hour.

Clearly in a k-dimensional schedule, the notion of optimality is simply to reduce k as much as possible (a quadratic schedule is faster than a cubic one, and so on), and this gives a simple greedy strategy for finding a multidimensional schedule for SURE, which is proven to be optimal.

- Try to determine a 1-dimensional schedule, using the linear programming formulation implied by Eqn. (28) or its generalization to variable-dependent, 1-dimensional schedules (not shown here). If the algorithm succeeds, we are done.

- If not, we seek to resolve a modified linear programming problem, where the inequalities are now non-strict (i.e., we seek scheduling hyperplanes, such that all dependences come



$$X[i,j] = f(X[i-1,j+1])$$
$$Y[i,j] = g(Y[i+1,j-1],X[i,j])$$

It can be easily verified from the EDG (shown left) of the above SURE that (i) the computations on each diagonal line are independent; (ii) the length of the longest path reaching a green or light gray node (resp. a red or dark gray node) at index point $[i,j]$ is i (resp. $i+2j+1$). The SURE thus does not admit a variable-independent (or even shifted linear) schedule. But clearly, a valid variable-dependent schedule is $t_X(i,j)=i$; $t_Y(i,j)=i+2j+1$.

Figure 5: Limitations of 1-dimensional schedules.

from to *either* a strictly preceding, *or even the same* hyperplane). If even the weak problem does not admit a feasible solution, the SURE does not admit a schedule, and we are done.

- Otherwise, we seek a solution where as many dependences as possible are satisfied in the strict sense (this is the greedy strategy).
- Next, we delete from the RDG, the edges corresponding to dependences that are strictly satisfied by the above solution
- We recursively seek a 1-dimensional schedule on each connected component of the resulting RDG (unless we have already recursed to a depth equal to the number of dimensions, in which case too, the SURE does not admit a schedule, and we are done).

6.4 Scheduling ARE's and SARE's

Let us now consider a single ARE:

$$\forall z \in D \quad X[z] = g(X[A_1 z + a_1] \ldots X[A_s z + a_s]) \tag{33}$$

Following the same arguments as for the case of a single URE, the scheduling constraints of Eqn. (26) reduce to: for $j = 1 \ldots s$

$$\begin{aligned} &\forall z \in D, \\ \Lambda z + \alpha + \Lambda' p &> \Lambda(A_j z + a_j) + \alpha + \Lambda' p \\ &\text{i.e., } \forall z \in D, \\ \Lambda((A_j - I)z + a_j) &< 0 \end{aligned} \tag{34}$$

Here, the z does not "cancel out" from the constraints, and thus we still have a potentially unbounded number of constraints to satisfy. However, we know that the domain, D is a polyhedron and admits a compact representation. In particular, we exploit the fact that an affine inequality constraint is satisfied at all points in a polyhedron iff it is satisfied at its extremal points. This leads to the following result.

Remark 3. *$\langle \Lambda, \alpha \rangle$ is a valid schedule for the* ARE *(33) if and only if for $j = 1 \ldots s$, for each vertex, σ and each ray, ρ of $\mathcal{D}$*

$$\begin{aligned} \Lambda\sigma + \alpha &> 0 \\ \Lambda\rho &\geq 0 \\ \Lambda\sigma &> \Lambda(A_j\sigma + a_j) \\ \Lambda\rho &\geq \Lambda A_j \rho \end{aligned}$$

Again, we have been able to exploit the compact representation (this time of the *domain* of the equation) to reduce the problem to the resolution of a fiite number of linear constraints. As with URE's and SURE's and using arguments analogous to those we have seen above, this result can be extended to shifted linear and variable-dependent (one- and multidimensional) schedules.

6.5 Undecidability results

We conclude this section with some important observations. The general problem of of determining a schedule for an SARE is undecidable. Indeed, there are many subtle related results.

- For a single URE defined either on an arbitrary polyhedral domain the scheduling problem is decidable.
- For an SURE defined over any *bounded* set of domains, the problem is decidable.
- For an SURE defined over an arbitrary set of domains, the problem is *undecidable*.
- For an SURE where all variables are defined over the entire positive orthant (a special case of unbounded domains), the problem is decidable.
- Since SARE's are more general thatn SURE's, the above results also hold for them. However, for an unbounded family of *parameterized* SARE's, each of which is defined over a bounded set of domains, the scheduling problem is also undecidable.

The algorithms that we have described in this section have nevertheless given necessary and *sufficient* conditions for determining schedules. This is consistent with the above results, since the scheduling algorithms restricted themselves to a specific class of schedules: affine schedules. This has an important sublte consequence on our parallelization problem. When we seek to compile an arbitrary Alpha program, we must account for the fact that the compiler may not be able to find such a schedule.

However, if the Alpha program (or equivalently the SARE) was the result of an exact data flow analysis of an ACL, we have an independent guarantee of the existence of a multidimensional affine schedule—the original (completely sequential) execution order of the ACL itself.

For Alpha, the operational semantics that we provided in the previous section gives us a "fall-back" strategy to compile programs that do not admit a multidimensional affine schedule, but an automatic parallelizer for ACL's does not need such a "fall-back" strategy.

7 Processor allocation in the polyhedral model

Analogous to the schedule which assigns a date to every operation (i.e., each instance of each variable in the SARE), a second key aspect of the parallelization is to assign a processor to each operation. This is done by means of a *processor allocation* function. For all the reasons already mentioned earlier, we insist that this function be determined only by analyzing the RDG, and that it be specified as a closed form function. As with schedules, we confine ourselves to *affine allocation* functions, defined as follows.

$$\mathrm{alloc}(\langle S_{\mathrm{X}}, z\rangle) = \mathcal{A}_{\mathrm{X}}(z) = \Phi_{\mathrm{X}} z + \phi_{\mathrm{X}} + \Phi'_{\mathrm{X}} p \tag{35}$$

where Φ_{Y} (resp. ϕ_{Y} and Φ'_{Y}) is an $a_{\mathrm{Y}} \times n_{\mathrm{Y}}$ (resp. $a_{\mathrm{Y}} \times 1$ and $a_{\mathrm{Y}} \times l$)integral matrix. Thus, the resulting image of D_{Y} by $\mathcal{M}_{\mathrm{Y}}$ is (contained in) an a_{Y} dimensional polyhedron.

We mention that there are is a difficulty with such allocation functions. In general, the number of processors that such functions imply is (usually approximated by) a polynomial function of the size parameters of the program (eg. one would need $\mathcal{O}(n^2)$ processors for matrix multiplication). This is usually too large to be used on a pragmatic machine, and we often treat the allocation function as the composition of two functions: an affine function that gives us the "virtual processors" and the second one being a mapping that specifies the emulation of these virtual processors by a physical machine (usually through directives that one finds in languages like HPF, such as the "block" and/or "cyclic" allocation of computations/data to processors). For the purpose of this section, we will focus on just the first part of the complete allocation function, namely the affine function. Obviously, this introduces certain limitations, which we will discuss in Section 10.

For an affine allocation function, observe that two points z and z' in D_{X} are mapped to the same processor if and only if $\Phi z = \Phi z'$, i.e., $(z - z')$ belongs to the null space or kernel[7] of Φ. Similarly, recall that two points z and z' in D_{X} are *scheduled* at the same time instant if and only if $z - z'$ is in the kernel of Λ_{X}.

Unlike the schedule, the allocation function does not have to satisfy any "causality" constraints, and hence there is considerable freedom in choosing it. Indeed, the only constraint that we must ensure is that it does not *conflict* with the schedule, in the sense that to oper-

[7]For any matrix M, its null space or kernel is $\mathrm{Ker}(M) \equiv \{z | Mz = 0\}$.

ations must not be scheduled simultaneously on the same processor. This can be formalized as follows.

Remark 4. *An allocation function as defined in Eqn. (35) is compatible with a schedule, as defined in Eqn. 25 if and only if*

$$\begin{aligned} &\forall z, z' \in D_X, \\ &\Phi_X z + \phi_X + \Phi'_X p = \Phi_X z' + \phi_X + \Phi'_X p \quad \Leftrightarrow \quad \Lambda_X z + \alpha_X + \Lambda'_X p \neq \Lambda_X z' + \alpha_X + \Lambda'_X p \end{aligned} \tag{36}$$

i.e.,

$$(z - z') \in \mathrm{Ker}(\Phi) \quad \Leftrightarrow \quad (z - z') \notin \mathrm{Ker}(\Lambda) \tag{37}$$

Recall that the *lineality space* of a polyhedron P is given by A if the *equality constraints* in the definition of P are of the form $Az = a$ for some constant vector a. Then the above constraint is equivalent to the following.

Remark 5. *For a variable whose domain, D_X has a lineality space given by A_X, an allocation function as defined in Eqn. (35) is compatible with a schedule as defined in Eqn. (25) if and only if the matrix $\begin{bmatrix} \Lambda_X \\ \Phi_X \\ A_X \end{bmatrix}$ is of full column rank.*

Once again, we have expressed the required constraints in a compact form independent of the size of the domains. Also note that we have a separate constraint for *each* variable, X, i.e., we assume that there is no possibility of conflict between the instances of different variables. This is because there are only finitely many variables in the SARE, and any potential conflict can be easily resolved by "serializing" among the (finitely many) conflicting operations.

In order to choose among the large number of potential allocation functions in the feasible space defined by the above constraints we may use two notions of *cost*. The first and natural one is the *number* of processors. This may be formalized as simply the number of integer points in the union of the images of each of the variable domains, D_X by the respective allocation functions, $\mathcal{A}_X$. Note that since this is in general a polynomial of the size parameters, we cannot use linear programming methods but have to take recourse of non-linear optimization. A second criterion is the communication engendered by the allocation (indeed, one often seeks to optimize this rather than the number of processors, since the latter will be later changed by the virtual-to-physical mapping). The polyhedral model provides us with a very clean way of reasoning about communication. Consider a (self) dependence $(z \rightarrow Az + a)$ in an SARE. Now, if the allocation function is given by 35, the computation $X[z]$ engenders a communication to processor $\mathcal{A}_X(Az + a)$ to $\mathcal{A}_X(z)$, i.e., a distance of $\Phi_X(z - Az - a)$. Note that for SURE's, this is a constant (since A is the identity). This provides us with a quantitative measure of the communication, and can be used to choose the allocation function optimally.

8 Memory allocation in the polyhedral model

The third key aspect of the static analysis of SARE's is the allocation of operations to memory locations. In this section we first introduce some basic machinery and then formulate the constraints that a memory allocation function must satisfy. We will use the forward substitution program (Figure 6) as a running example. As with the schedule and the processor allocation function, the *memory allocation*, is also an affine function, defined as follows.

$$\mathrm{Mem}(\langle S_{\mathrm{Y}}, z\rangle) = \mathcal{M}_{\mathrm{Y}}(z) = \Pi_{\mathrm{Y}} z + \pi_{\mathrm{Y}} + \Pi'_{\mathrm{Y}} p \tag{38}$$

Here, Π_{Y} (respectively π_{Y} and Π'_{Y}) is an $(n_{\mathrm{Y}} - m_{\mathrm{Y}}) \times n_{\mathrm{Y}}$ (respectively $(n_{\mathrm{Y}} - m_{\mathrm{Y}}) \times 1$ and $(n_{\mathrm{Y}} - m_{\mathrm{Y}}) \times l$) integral matrix. Thus, the resulting image of D_{Y} by $\mathcal{M}_{\mathrm{Y}}$ is (contained in) an $(n_{\mathrm{Y}} - m_{\mathrm{Y}})$ dimensional polyhedron, i.e., m_{Y} dimensions are "projected out". For simplicity in the analysis presented here, we assume henceforth that m_{Y} and M'_{Y} are both 0.

As with the processor allocation, a memory allocation function is characterized by null space of Π_{Y}, and can be completely specified by means of m_{Y} constant vectors, ρ_i for $i = 1 \ldots m_{\mathrm{Y}}$, that form a basis for $\mathrm{Ker}(M)$, and which can be unimodularly completed (i.e., there exists an $n_{\mathrm{Y}} \times n_{\mathrm{Y}}$ unimodular[8] matrix, whose first m_{Y} columns are $\rho_1 \ldots \rho_{m_{\mathrm{Y}}}$). One can visualize that an index point z is mapped to the same memory location as $z + \sum_i \mu_i \rho_i$, for any integer linear combination of the ρ_i's.

Example: For the forward substitution program (Figure 6) we may propose

[8]A square integer matrix M is said to be unimodular if and only if $\det(M) = \pm 1$. Hence its inverse exists and is integral.

```
system ForwardSubstitution : { N | N>1 }
              ( A : { i,j | 0<j<i<=N } of real;
                B : { i   | 0<i<=N } of real)
       returns ( X : { i   | 0<i<=N } of real );
var
  f : {i,j | (2,j+1)<=i<=N; 0<=j} of real;
let
  f[i,j] = case
        {| j=0} : 0[];
        {| 1<=j} : f[i,j-1] + A * X[j];
           esac;
  X[i] = case
        {| 2<=i} : B[i] - f[i,i-1];
        {| i=1} : B[i];
         esac;
tel;
```

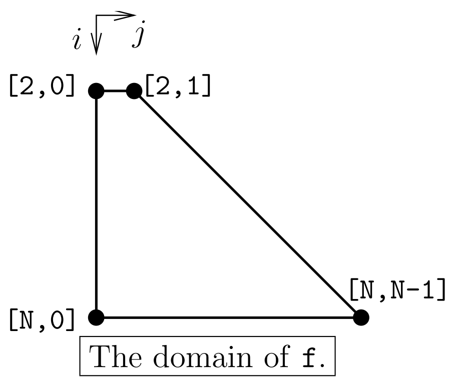


Figure 6: Forward substitution program with reduction replaced by a series of binary additions.

that the variable `f` be allocated to a one-dimensional memory by the projection $M_{\texttt{f}} = [1, -1]$, which allocates `f[i,j]` to memory location `i-j`. This allocation function is specified by the projection vector $\rho_{\texttt{f}} = [1, 1]$. The image of the domain of `f` by the projection is `{m | 1 <= m <= N}`, i.e., a vector of length `N`.

Alternatively, we may propose that all the points in the domain of `f` be allocated to a single scalar, i.e., a projection of its domain to $\mathcal{Z}^0$. Here the projection vectors are the two unit vectors $[0, 1]$ and $[1, 0]$.

Observe that the choice of the projection vectors is not unique. In the first case, $[-1, 1]$ is also a valid projection, and in the second case the columns of *any* 2×2 unimodular matrix are valid projections. *end of example.*

We now study the validity of memory allocation functions. Since the memory allocation is in general, a many-to-one mapping, certain (in fact, most) values will be overwritten as the computation proceeds. We need to ensure that no value is overwritten before all the computations that depend on it are themselves executed, as formalized below.

Definition 5. *A memory function* Mem_{Y} *is valid if and only if for all* $z \in D_{\text{Y}}$*, the* ***next write*** *after* $t_{\text{Y}}(z)$ *into the memory location* $\text{Mem}_{\text{Y}}(z)$ *occurs after* ***all uses*** *of* $\text{Y}[z]$*.*

Note that the above definition uses the term "after", which clearly depends on the schedule. Thus, the memory allocation function, unlike the processor allocation has an intricate interaction with the schedule. Nevertheless, we have a condition analogous to the one for the processor allocation, but which give us only necessary conditions.

Remark 6. *For a variable whose domain, D_{Y} has a lineality space given by A_{Y}, a memory allocation function as defined in Eqn. (38) is valid if the matrix* $\begin{bmatrix} \Lambda_{\mathrm{Y}} \\ \Pi_{\mathrm{Y}} \\ A_{\mathrm{Y}} \end{bmatrix}$ *is of full column rank.*

We also assume that for $i = 1 \cdots m_{\mathrm{Y}}$, the vectors ρ_i's are such that $\Lambda_{\mathrm{Y}}\rho_i$ is lexicographically positive. Note that this does not impose any loss of generality: we may always choose the sign of the ρ_i's appropriately.

In the remainder of this section, we first formalize three notions—the next-write function, the usage set, and the lifetime—using which we formulate the validity constraints that the memory allocation function must satisfy.

8.1 Next-write function

Now consider, for any $z \in D_{\mathrm{Y}}$, the *next* point, z', that overwrites the memory location of $Y[z]$. We want to express z' as a function of z.

Definition 6. *For any point $z \in D_{\mathrm{Y}}$, the* ***next write*** *is the earliest scheduled point that satisfies the following constraints:*

$$\left\| \begin{array}{lcl} \Pi_{\mathrm{Y}} z & = & \Pi_{\mathrm{Y}} z' \\ z & \in & D_{\mathrm{Y}} \\ z' & \in & D_{\mathrm{Y}} \\ \Lambda_{\mathrm{Y}} z & \prec & \Lambda_{\mathrm{Y}} z' \end{array} \right. \qquad (39)$$

We write it as $z + \sigma_{\mathrm{Y}}(z)$. We define $\Lambda_{\mathrm{Y}}\sigma_{\mathrm{Y}}(z)$ which is the time interval between the computation of $Y[z]$ and its destruction as the ***overwrite window*** *of $Y[z]$.*

Note that the constraints (39) define a set of k disjoint polyhedra, parameterized by z, and hence the next write can be obtained by resolving k parametric integer programming problems [15]. Hence, $\sigma_{\mathrm{Y}}(z)$ is a piece-wise affine function of z.

Now, observe that if we drop the constraints that z and z' must belong to D_{Y} from (39), the solution is simply $z + \sigma_{\mathrm{Y}}$ for some *constant* vector, σ_{Y}, which we call the *next-write vector* for Y. It is clear that $\Lambda_{\mathrm{Y}}\sigma_{\mathrm{Y}}$ is a lower bound on the overwrite window of $Y[z]$, and for points "far" from the boundaries, this approximation is exact. Furthermore, since our domains are large enough, such points exist (indeed, are the rule rather than the exception). The above ideas are formulated in the proposition below and illustrated through the subsequent examples.

Proposition 1. *$\sigma_{\mathrm{Y}}(z)$ is a piece-wise affine function, such that at points sufficiently far from the boundaries of D_{Y} its value is a constant, σ_{Y}*

Example: For the forward substitution program (Figure 6), let the memory allocation function for `f` be specified by $\Pi_{\mathtt{f}} = [1, -1]$, and the schedule be given by $t_{\mathtt{f}}(i,j) = \begin{pmatrix} i+j \\ j \end{pmatrix}$, i.e., $\Lambda_{\mathtt{f}} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, $\alpha_{\mathtt{f}} = 0$. Recall that the schedule may be visualized as follows: the first row represents the family of lines $i + j = h$ corresponding to the first time dimension (i.e., the h-th "hour"), and the second row specifies the "minutes" ($j = m$) within each hour, i.e., the second time dimension. Then, for a point $z \in D_{\mathtt{f}}$ the next write into the same memory location as z is $\mathtt{f}(z')$, where

$$z' = \begin{cases} \{i,j \mid 1 < i < N; 0 \le j < i\} & : \quad z + \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ \{i,j \mid i = N; 0 \le j < i\} & : \quad \text{undefined.} \end{cases}$$

On the other hand, if (for the same schedule), the memory allocation is specified by the two unit vectors (i.e., all the points in the domain of `f` are allocated to a single scalar), then

$$z' = \begin{cases} \{i,j|2 < i \le N; 0 \le j < i\} & : \; z + \begin{pmatrix} -1 \\ 1 \end{pmatrix} \\ \{i,j|i = 2; j = 0\} & : \; z + \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ \{i,j|2 < i; j = i-1; 2i-1 \le N\} & : \; z + \begin{pmatrix} i \\ -i+1 \end{pmatrix} \\ \{i,j|2 < i; j = i-2; i+j \le N-1\} & : \; z + \begin{pmatrix} j+1 \\ 0 \end{pmatrix} \\ \{i,j|2 < i; j = i-2; i+j \ge N\} & : \; z + \begin{pmatrix} 0 \\ i-N+1 \end{pmatrix} \\ \{i,j|j = i-1 = N-1\} & : \; \text{undefined.} \end{cases}$$

Observe how $z' - z = \sigma_{\mathtt{f}}(z)$ is a piece-wise affine function of i and j, and note how the first clause corresponds to interior points while all the other clauses are boundary cases. They occur on subdomains with equalities, and it can be verified that, for each $z \in D_{\mathtt{f}}$, there does not exist $z'' \in D_{\mathtt{f}}$ such that

$$\Lambda_{\mathtt{f}}(z + \begin{pmatrix} -1 \\ 1 \end{pmatrix}) \prec \Lambda_{\mathtt{f}} z'' \prec \Lambda_{\mathtt{f}} z'.$$

In other words, the value of $\sigma_{\mathtt{f}}$ due to the first clause is a lower bound on those predicted by the other clauses. Thus, for the two allocation functions, the *next-write vector* $\sigma_{\mathtt{f}}$, is respectively $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ and $\begin{pmatrix} -1 \\ 1 \end{pmatrix}$. *end of example.*

8.2 Usage set

Assume that the (sub) expression $Y.(z \to Mz + m)$ occurs on the right-hand side of the equation defining X. The *context* where it appears in the Alpha program (e.g., within a case and/or a restriction) defines a domain $D'_{\mathrm{X}} \subseteq D_{\mathrm{X}}$ where the dependency is said to hold. This information can be readily computed from the program text and is written as follows

(F serves to name the dependency):

$$F \ : \ \forall z \in D'_{\mathrm{X}}, \quad \mathrm{X}[z] \to \mathrm{Y}[Mz+m] \tag{40}$$

Let us also use $\mathcal{F}_Y$ to denote the *set* of dependencies on Y, i.e., the set of dependencies where the variable on the right-hand side is Y. For any $z \in D_{\mathrm{Y}}$, we are interested in determining the users of $Y[z]$, with respect to a dependency, F. Note that since the affine function $Mz+m$ may not be invertible, there may be multiple users. They constitute some subset of D_X (or more precisely, of D'_X, since the dependency holds only there). Let us call this set D^F_X. The following properties must be satisfied:

$$\left\| \begin{array}{lcl} z & = & Mz'+m \\ z & \in & D_{\mathrm{Y}} \\ z' & \in & D'_{\mathrm{X}} \end{array} \right. \tag{41}$$

This can be viewed as the set of points $\begin{pmatrix} z \\ z' \end{pmatrix}$ belonging to a polyhedron. Alternatively, and equivalently, it can also be viewed as a set of points z' belonging to a polyhedron *parameterized* by z (and of course the system parameters), denoted by $D^F_{\mathrm{X}}(z)$. It is this interpretation that we shall use.

Example: For the forward substitution program (Figure 6), the dependencies on variable $\mathtt{f}$ are

$$\begin{array}{lll} F_1: \ \{i,j,N \mid 1 \le j < i \le N\} & : & \mathtt{f}[i,j] \to \mathtt{f}[i,j-1] \\ F_2: \ \{i,N \mid 2 \le i \le N\} & : & \mathtt{X}[i] \to \mathtt{f}[i,i-1]. \end{array}$$

Consider the first dependency, F_1. For any point $z = \begin{pmatrix} i \\ j \end{pmatrix} \in D_{\mathtt{f}}$, the set of its users $z' = \begin{pmatrix} i' \\ j' \end{pmatrix}$ with respect to F_1, is

$$\begin{aligned} D_{\mathtt{f}}^{F_1}(z) &= \{i', j', i, j, N \mid i = i', j' = j + 1\} \\ &\quad \cap \{i', j', i, j, N \mid 2 \le i \le N; 0 \le j < i\} \\ &\quad \cap \{i', j', i, j, N \mid 1 \le j' < i' \le N\} \\ &= \{i', j', i, j, N \mid 2 \le i' = i \le N; j' = j + 1; 0 \le j < i - 1\}. \end{aligned}$$

This is viewed as a two-dimensional polyhedron over i' and j', parameterized by i, j (and N). Observe that the very last constraint above, namely $j < i - 1$, correctly implies that points on the $i = j + 1$ are not used by any computation (with respect to the dependency F_1).

For the dependency, F_2, we proceed similarly (recall that we are considering usage by `X`, so z is two-dimensional, and z' is one-dimensional).

$$\begin{aligned} D_{\mathtt{f}}^{F_2}(z) &= \{i', i, j, N \mid i = i', j = i' - 1\} \\ &\quad \cap \{i', i, j, N \mid 2 \le i \le N; 0 \le j < i\} \\ &\quad \cap \{i', i, j, N \mid 2 \le i' \le N\} \\ &= \{i', i, j, N \mid i = i' = j + 1; 2 \le i \le N\}. \end{aligned}$$

Observe again, that only values of `f` computed on the $i = j + 1$ boundary are used (with respect to this dependency), and this is correctly predicted by our computation of $D_{\mathtt{f}}^{F_2}(z)$. Furthermore, the set of points that use `f[i,i-1]` is a singleton in the domain of `X`, namely `X[i']` where `i = i'`.

To illustrate the case when the dependency function is not invertible, consider the following dependency on `X` (though, `X` being an output variable, is not a candidate for memory reuse).

$$F_3: \{i, j, N \mid 1 \le j < i \le N\} \; : \; \mathtt{f}[i, j] \to \mathtt{X}[j]$$

Here, $z = (i)$ is one-dimensional, and $z' = \begin{pmatrix} i' \\ j' \end{pmatrix}$ is two-dimensional, and we have

$$\begin{aligned} D_{\mathtt{X}}^{F_3}(z) &= \{i', j', i, N \mid i = j'\} \\ &\quad \cap \{i', j', i, N \mid 1 \leq i \leq N\} \\ &\quad \cap \{i', j', i, N \mid 1 \leq j' < i' \leq N\} \\ &= \{i', j', i, N \mid j' = i; 1 \leq i < N; i < i' \leq N\}. \end{aligned}$$

Observe again, that this correctly predicts that the first $N-1$ values of `X` are used. Viewing this as a family of two-dimensional polyhedra parameterized by i (and N) we see that a given `X[i]` is used by multiple points `f[i', j']` such that $j' = i$ and $i < i' \leq N$. *end of example.*

8.3 Lifetime

Since we are given a schedule for the program, we know that for any $z \in D_{\mathrm{Y}}$, $Y[z]$ is computed at (the k-dimensional) time instant $\Lambda_{\mathrm{Y}} z + \alpha_{\mathrm{Y}}$, and for any $z' \in D_{\mathrm{X}}^F(z)$, $X[z']$ is computed at time $\Lambda_{\mathrm{X}} z' + \alpha_{\mathrm{X}}$. Hence the time between the production of $Y[z]$ and its use by $X[z']$ is simply $\Lambda_{\mathrm{X}} z' + \alpha_{\mathrm{X}} - \Lambda_{\mathrm{Y}} z - \alpha_{\mathrm{Y}}$. We have the following definition, where $\mathrm{Lmax}_{x \in S} f(x)$ denotes the lexicographic maximum of $f(x)$ over the set S.

Definition 7. *The* partial lifetime*, $d_{\mathrm{Y}}^F(z)$, of $Y[z]$ with respect to the dependency F is*

$$d_{\mathrm{Y}}^F(z) = \underset{z' \in D_{\mathrm{X}}^F(z)}{\mathrm{Lmax}} (\Lambda_{\mathrm{X}} z' + \alpha_{\mathrm{X}} - \Lambda_{\mathrm{Y}} z - \alpha_{\mathrm{Y}}). \tag{42}$$

The (total) lifetime *of $Y[z]$ is*

$$d_{\mathrm{Y}}(z) = \underset{F \in \mathcal{F}_Y}{\mathrm{Lmax}}\, d_{\mathrm{Y}}^F(z). \tag{43}$$

The lifetime of the entire variable Y is

$$d_{\mathrm{Y}} = \underset{z \in D_{\mathrm{Y}}}{\mathrm{Lmax}}\, d_{\mathrm{Y}}(z). \tag{44}$$

We observe that $d_{\mathrm{Y}}^F(z)$, being the lexicographic maximum over a polyhedron parameterized by z, is a piece-wise affine function of z (and the system parameters). Furthermore,

$d_{\mathrm{Y}}(z)$, being the lexicographic maximum of a finite number of such functions (note that $\mathcal{F}_{\mathrm{Y}}$ is a finite set of dependencies), is also a piece-wise affine function of z. Hence, d_{Y} is the lexicographic maximum, not of an affine, but a *piece-wise affine*, cost function over the domain D_{Y}. This can be expressed as the resolution of a finite number of parametric integer programming problems (one for each of the "pieces" of the cost function), and hence, is the lexicographic maximum of a finite number of piece-wise affine functions. As a result, we have the following.

Proposition 2. *$d_{\mathrm{Y}}(z)$ is a piece-wise affine function of z and the system parameters.*

Example: Continuing with the forward substitution program, let the respective schedules for `f` and `X` be given by $t_{\mathtt{f}}(i,j) = i+j$ and $t_{\mathtt{X}}(i) = 2i$. The lifetime of any $\mathtt{f}(i,j)$ is especially easy to determine because our usage sets are singletons. Since $\mathtt{f}(i,j)$ is computed at date $i+j$, its partial lifetime with respect to F_1 is the lexicographic maximum of $t_{\mathtt{f}}(i',j') - t_{\mathtt{f}}(i,j)$ over $D_{\mathtt{f}}^{F_1}(z)$, and since $i = i'$ and $j' = j+1$,

$$d_{\mathtt{f}}^{F_1}(z) = 1.$$

Similarly, for F_2 the usage sets are again singletons, and it is easy to see that $\mathtt{f[i,j]}$ for $2 \leq i = j+1 \leq N$ is used precisely by $\mathtt{X[i']}$, where $i' = i$. These points are respectively computed at $i+j$ and $2i'$, whose difference is always 1. Hence

$$d_{\mathtt{f}}^{F_2}(z) = 1.$$

If we choose the alternative (sequential) schedule given by $t_{\mathtt{f}}(i,j) = \begin{pmatrix} i+j \\ j \end{pmatrix}$ and $t_{\mathtt{X}}(i) = \begin{pmatrix} 2i \\ i \end{pmatrix}$, the lifetime is now a two-dimensional function, and can be easily shown to be

$$\begin{aligned}
d_{\mathtt{f}}^{F_1}(z) &= \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} (z + \begin{pmatrix} 0 \\ 1 \end{pmatrix}) - \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} z \\
&= \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\
d_{\mathtt{f}}^{F_2}(z) &= \begin{pmatrix} 2 \\ 1 \end{pmatrix} z' - \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} z \\
&= \begin{pmatrix} 2 \\ 1 \end{pmatrix} (i) - \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \\
&= \begin{pmatrix} i - j \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.
\end{aligned}$$

Note that the final simplification is possible because $D_{\mathtt{f}}^{F_2}(z)$ is nonempty only when $i = j + 1$. *end of example.*

8.4 Validity conditions for the memory allocation function

We now develop necessary and sufficient conditions on the vectors ρ_i's so that the basic constraint on memory allocation function (Definition 5) holds. Recall that

- the computation that overwrites $Y[z]$ is $Y[z + \sigma_{\mathrm{Y}}(z)]$;
- the set of users of $Y[z]$ with respect to a dependency, F, is $D_{\mathrm{X}}^{F}(z)$;
- Y$[z]$ is computed at time $\Lambda_{\mathrm{Y}} z + \alpha_{\mathrm{Y}}$;
- $Y[z + \sigma_{\mathrm{Y}}(z)]$ is computed at time $\Lambda_{\mathrm{Y}}(z + \sigma_{\mathrm{Y}}(z)) + \alpha_{\mathrm{Y}}$; and
- for any $z'' \in D_{\mathrm{X}}^{F}(z)$, $X[z'']$ is computed at time $\Lambda_{\mathrm{X}} z'' + \alpha_{\mathrm{X}}$.

Hence the condition imposed by Definition 5 will hold *in the context of* F if and only if

$$\forall z \in D_{\mathrm{Y}} \text{ and } \forall z'' \in D_{\mathrm{X}}^{F}(z), \quad \Lambda_{\mathrm{X}} z'' + \alpha_{\mathrm{X}} - \Lambda_{\mathrm{Y}} z - \alpha_{\mathrm{Y}} \preceq \Lambda_{\mathrm{Y}} \sigma_{\mathrm{Y}}(z) \qquad (45)$$

Note that this inequality is nonstrict—we allow $Y[z]$ to be read at the "same" instant that its memory location is overwritten. This assumes that the machine architecture provides such synchronization. If this is not the case, we could insist on strict inequality, without changing the nature of the analysis.

Since the size of the sets D_{Y} and $D_{\mathrm{X}}^{F}(z)$ may be arbitrary, there may be an unbounded number of constraints to be satisfied. However, the fact that $D_{\mathrm{X}}^{F}(z)$ is a polyhedron allows us to exploit the power of the polyhedral model, namely that (45) holds at *all* $z'' \in D_{\mathrm{X}}^{F}(z)$ if and only if it is satisfied by the points of $D_{\mathrm{X}}^{F}(z)$ that maximize, in the lexicographic order, $\Lambda_{\mathrm{X}} z''$. Using Definition 7, this reduces to

$$\forall z \in D_{\mathrm{Y}},\ d_{\mathrm{Y}}^{F}(z) \preceq \Lambda_{\mathrm{Y}} \sigma_{\mathrm{Y}}(z). \tag{46}$$

This defines the constraints on the overwriting of $Y[z]$ with respect to a single dependency. In order to determine the necessary and sufficient conditions on the memory allocation function, we need to extend the above analysis to $\mathcal{F}_{Y}$. Hence, a memory allocation function is valid if and only if

$$\forall z \in D_{\mathrm{Y}},\ d_{\mathrm{Y}}(z) \preceq \Lambda_{\mathrm{Y}} \sigma_{\mathrm{Y}}(z). \tag{47}$$

Now, observe that $d_{\mathrm{Y}}(z)$ is a piece-wise affine function of z, and hence, we can separate (47) into a finite number of similar constraints over disjoint subsets, D_{Y}^{i}, of D_{Y}, each of which has a single affine function $d_{\mathrm{Y}}^{i}(z)$, on the left. As a result, we can seek a separate memory allocation function for each subdomain, D_{Y}^{i}. Our experience has shown that piece-wise-linear allocations are often very useful (only certain subdomains may need larger memory, while for others there may be additional projection dimensions). Now, since we could also use a single homogeneous allocation function over the entire domain, D_{Y}, by ensuring that $d_{\mathrm{Y}} \preceq \Lambda_{\mathrm{Y}} \sigma_{\mathrm{Y}}$, which is a safe approximation, the improvement due to piece-wise-linear allocations comes at a price of some overhead, and increased compilation time. Our current implementation uses a homogeneous allocation by default and performs the refined analysis as a user-specified option.

Like the lifetime function, $\sigma_{\mathrm{Y}}(z)$ is also a piece-wise affine function of z. Moreover, $\sigma_{\mathrm{Y}}(z)$ is a constant (see Proposition 1) almost everywhere except near the boundaries of D_{Y}, and hence of the corresponding D^i_{Y}'s. This constant (namely, the next-write vector, σ_{Y}) is a lower bound on $\sigma_{\mathrm{y}}(z)$. We will therefore approximate $\sigma_{\mathrm{Y}}(z)$ by σ_{Y} and henceforth use the following sufficient condition.

$$\forall z \in D^i_{\mathrm{Y}},\ d_{\mathrm{Y}}(z) \preceq \Lambda_{\mathrm{Y}}\sigma_{\mathrm{Y}}. \tag{48}$$

Finally, we again use the key idea of the polyhedral model, namely that (48) holds for all points $z \in D^i_{\mathrm{Y}}$ if and only if it holds for the points in D^i_{Y} which maximize the left-hand side, i.e., if and only if

$$d^i_{\mathrm{Y}} \preceq \Lambda_{\mathrm{Y}}\sigma_{\mathrm{Y}}, \tag{49}$$

where d^i_{Y} is the lifetime of Y over the subdomain D^i_{Y}.

Hence we have reduced the validity conditions for a memory allocation function to the satisfaction of a finite number of linear constraints. The following proposition relates this constraint on σ_{Y} to the ρ_i's.

Proposition 3. *A set of projection vectors (ρ_i) defines a valid memory projection for a variable* Y *if and only if for all integral linear combinations η, of ρ_i's*

$$0 \prec \Lambda_{\mathrm{Y}}\eta \Rightarrow d_{\mathrm{Y}} \preceq \Lambda_{\mathrm{Y}}\eta.$$

Proof.

If part. If the ρ_i's are such that (49) holds $\forall z \in \mathbb{Z}^{n_{\mathrm{Y}}}$, let if possible $\eta = \sum_{i=1}^{m_{\mathrm{Y}}} \mu_i\rho_i$, $\mu_i \in \mathbb{Z}$ such that $0 \prec \Lambda_{\mathrm{Y}}\eta$ and $d_{\mathrm{Y}} \not\preceq \Lambda_{\mathrm{Y}}\eta$. Then, for some $z \in \mathbb{Z}^{n_{\mathrm{Y}}}$, $0 \prec \Lambda_{\mathrm{Y}}\eta \prec d_{\mathrm{Y}} \preceq \Lambda_{\mathrm{Y}}\sigma_{\mathrm{Y}}$. This implies that η and σ_{Y} are distinct, and hence it is $Y[z+\eta]$ and not $Y[z+\sigma_{\mathrm{Y}}]$ which is the next write into the same memory cell as $Y[z]$, a contradiction.

Only if part. Obvious, since σ_{Y} is a linear combination of the ρ_i's such that $0 \prec \Lambda_{\mathrm{Y}}\sigma_{\mathrm{Y}}$. $\square$

Finally, as with the memory and the processor allocation functions, there is a well-defined notion of optimality for the memory allocation, namely the volume of memory that is used for a given schedule. Indeed, since this is a polynomial function of the size parameters, the most important criterion to minimize is the *number* of linearly independent projection vectors, ρ_i. It can be shown (constructively) that this is equal to 1 plus the number of leading zeroes in d_Y, the lifetime vector for the variable.

9 Code generation

In the previous three sections we have addressed some essential issues of *analyzing* SARE's. Specifically, we have given methods to determine three types of functions: *schedules* to assign a date to each operation, *processor allocation* to assign operations to processors, and *memory allocations* to store intermediate results. These functions may be chosen so as to optimize certain criteria under certain assumptions, but note that the optimization problems are far from being resolved, and indeed, there are many interesting open problems. Our focus in this section is to consider a different problem orthogonal to the choice of these functions. This is the problem of code generation—given the above three functions, how do we produce parallel code that "implements" these choices.

By insisting that code generation is independent of the choice of the schedule and allocation functions, we achieve a separation of concerns—for example, the methods described here may be used to produce either sequential or parallel code for programmable (i.e., instruction-set) processors, or even VHDL descriptions of application-specific or non-programmable hardware that implements the computation specified by the SARE.

We first show how to implement or compile an SARE (ALPHA program) in the *complete absence* of any static analysis. In other words, we present a simple operational semantics for ALPHA and describe how to produce naive code. This highlights the separation of concerns mentioned above, and also provides us with a baseline, fall-back implementation, which we are able to progressively improve, as and when static analysis is available. Next, we shall see how the schedule (plus the processor allocation if available) can be used to enable us to generate efficient imperative but memory inefficient code, and finally, we describe how the memory allocation function can be used to perform very simple and minor modifications of this code to produce memory-efficient code.

9.1 Polyhedron scanning

Before we enter into the details, we develop an important tool that we need, namely a resolution of the *polyhedron scanning* problem, defined as follows.

Problem: Given a (possibly parameterized) polyhedron, $\mathcal{P}$ construct a loop nest to visit the integral points in $\mathcal{P}$ in lexicographic order of the indices.

This problem has been well studied by parallelizing compiler researchers, and the solution is based on the well-known technique of Fourier-Motzkin elimination. This can be done by what is called **separation of polyhedra** and the key step here involves *projecting* a $k+1$-dimensional polyhedron, $P(z_1, \ldots z_k, z_{k+1}) \equiv \{z | Az \geq b\}$ onto its first k indices. In order to do this, we partition each row of A into one of three categories: those that (i) are independent of, (ii) provide a lower bound on, or (iii) provide an upper bound on z_{k+1}. From every pair of upper and lower bound inequalities, we eliminate z_{k+1} to get a new inequality that does not include z_{k+1}, and to this list we add the ones from the first category. This yields an alternative and equivalent representation of the polyhedron, P. By successively eliminating z_k, $z_{k-1} \ldots$ we may rewrite the constraints of the polyhedron in a sequence where the indices are "separated". As an example, consider the polyhedron, `{i,j,k| i>=0; -1+M>=0; -j+N>=0; k>=0; i+j-k>=0}`, in the context of parameters constrained by, `{N,M| N,M>0}`, and let us seek to visit all its integer point in the "scanning order", `{j,k,i}`. Polyhedron separation in this order is achieved by first eliminating `i`, then `k` and finally `j`. This yields the format shown on the left below, from which, we can construct the loop nest shown alongside, by simply "pretty-printing" it—each line introduces a new index, and its lower and upper bounds are evident from the inequalities on the line.

```
{j| 0<=j<=N} ::                        for (j=0; j++; j<=N)
  {k,j| 0<=k<=j+M} ::                    for (k=0; k++; k<=j+M)
    {i,k,j| 0<=i<=M; i>=k-j}               for (i=max(0,k-j); i++; i<=M)
```

If the dual (generator) representation of the polyhedron is available, many optimizations can be performed in this basic step (instead of pairwise combination of all the lower and upper bound constraints during each projection step, we can simply add a pair of rays to the generator representation).

It is important to note here that the Fourier-Motzkin elimination method is *exact* for real and rational polyhedra, but *non-exact* for integral polyhedra. This is because in general, the projection of an integral polyhedron is not necessarily also in integral polyhedron. However, there are well known methods to make the method exact for integral polyhedra. Polyhedron scanning is an important component of code generation.

9.2 Alpha operational semantics: naive code generation

In contrast with the denotational semantics given in Section 5, we now provide a simple operational semantics, one that allow us to develop our fall-back strategy for executing Alpha programs in the absence of *any* static analysis.

Semantics of expressions: Recall that since Alpha expressions denote mappings from indices to values, and operational semantics may be developed if we first define this mapping. We do so as a simple interpreter, in terms of a function Eval : $\langle\texttt{Exp}\rangle \times \mathcal{Z}^n \rightarrow$ Type. We first introduce a mechanism for computing and propagating "errors" (although this is not strictly necessary).

$$\text{Eval}(\langle\texttt{exp}\rangle, z) = \begin{cases} \text{Eval}'(z) & \text{if } z \in \text{Dom}(\texttt{exp}) \\ \perp & \text{otherwise} \end{cases}$$

Eval' is defined recursively, with six cases corresponding to the six syntax rules (see Table 1) for Alpha expressions.

$$\begin{aligned}
\text{Eval}'(\langle\texttt{Const}\rangle, z) &= C \\
\text{Eval}'(\langle\texttt{E1}\rangle\ \texttt{op}\ \langle\texttt{E2}\rangle, z) &= \text{Eval}'(\langle\texttt{E1}\rangle, z) \oplus \text{Eval}'(\langle\texttt{E2}\rangle, z) \\
\text{Eval}'(\texttt{case} \ldots \langle\texttt{Ei}\rangle \ldots \texttt{esac}, z) &= \begin{cases} \quad\vdots & \\ \text{Eval}'(\langle\texttt{Ei}\rangle, z) & \text{if } z \in \text{Dom}(\langle\texttt{Ei}\rangle) \\ \quad\vdots & \end{cases} \\
\text{Eval}'(\texttt{D} : \langle\texttt{E}\rangle, z) &= \text{Eval}'(\langle\texttt{E}\rangle, z) \\
\text{Eval}'(\langle\texttt{E}\rangle.f, z) &= \text{Eval}'(\langle\texttt{E}\rangle, f(z)) \\
\text{Eval}'(\langle\texttt{Var}\rangle, z) &= \text{EvalVar}(z)
\end{aligned}$$

Semantics of equations: The *denotational semantics* of Alpha equations essentially specify that equations denote "additions" to a "store of definitions" (such semantics are fairly standard and hence were not described in Section 5). The corresponding operational semantics are also straightforward,namely that every equation causes a function to be defined and added to the store, and the function body is the operational semantics of the expression on the rhs of the equation.

$$\text{Eval}(\texttt{Var = }\langle\texttt{Exp}\rangle) = (\text{defun EvalVar}(\text{Eval}(\langle\texttt{Exp}\rangle z)$$

Semantics of programs ALPHA programs are systems, and they *denote* mappings from input variables to output variables. The corresponding operational semantics are also straightforward. They require a mechanism to specify the following steps:

1. Read Input Variables
2. Compute (Local) and Output Variables
3. Write Output Variables

This can be achieved by a simple set of loops to scan the domains of each output variable, `Var`, and at each point, call the function `EvalVar`.

This essentially produces an interpreter, and suffers from the standard drawbacks of interpretive implementations of functional languages, namely the recomputation of previously evaluated values (for example, a program to compute the n-th Fibonacci number will take time exponential in n).

However, we may modify the operational semantics to exploit a well-known strategy used in functional languages, namely "caching" or tabulation. Instead of recomputing previously evaluated values, we "cache" them by storing them in a table (the Eval$'$ function is modified so that instead of making a recursive call, it first checks if the value has been previously evaluated and stored. he recursive call is made only if this is not so, and when it returns the value is "cached" into the table. In ALPHA this requires a very simple modification, namely the allocation of memory for the tables. This memory corresponds to the size of the domains and can be allocated as an array. The resulting code has the following structure:

- Declarations of multidimensional array variables corresponding to each local and output variable in the program.

- Definition of the functions EvalVar for all variables. For the local and output variables this follows the semantics as defined above. For input variables, such a function simply reads the input array at the specified address.
- A main program consisting of one set of loops (produced by polyhedron-scanning) that visit the domain of each of the output variables, and call the corresponding EvalVar function at each point (the order of each of the output variables, as well the order in which the domains are scanned is immaterial).

Finally we point out that there is very simple and implicit parallelism, and the code may be very easily be modified to run in a demand-driven manner on a multi-threaded machine—all the pointwise operators are strict, and this is where a synchronization is necessary to ensure that arguments are available before they are combined to produce results; everything else including the function calls, can be done in parallel.

9.3 Exploiting static analysis: imperative code generation

The code produced in the absence of static analysis suffers from two main drawbacks. Since there is a function call for each evaluation, there is a considerable overhead of context switching. Furthermore, the memory allocated to each domain corresponds to the entire domain, and this code is memory inefficient. We will now see how the results of static analysis, namely the schedule and the processor and memory allocation functions can be used to improve the code. The main idea is summarized as follows.

- We rewrite the SARE, by means of an appropriate CoB's, such that the first k indices of the domains of *all* variables are the k time dimensions; the remaining indices are to be interpreted as "virtual processors".
- We produce declarations of multidimensional arrays to store the variables of the program. For each variable, we allocate memory corresponding to the smallest rectangular box enclosing its domain.
- We next generate code that visits the union of the domains of all the variable in the lexicographic order imposed by the fist k indices (loops scanning other indices, if any, are annotated as `forall` loops). At each point visited, we simply evaluate the body

of the SARE *without* recursive calls, and without testing whether the arguments have been previously evaluated (the schedule guarantees that this is unnecessary).

- Note that the code that we produce here is *single assignment* form: each memory location (representing an *operation* in the original program) is stored in a distinct memory location. Now, suppose that, in addition to the schedule and the processor allocation, we also have for each variable of the SARE, a memory allocation function, $\mathcal{M}_i$ which gives a valid address that the result of $S_i[z]$ is stored at memory address $\mathcal{M}_i(z)$. Then we may easily generate *multiple assignment* code by modifying the above code as follows.
  - Modify the preamble so that it allocates memory arrays for only (the bounding box of) $\mathcal{M}_i(D_i)$, the *image* of D_i by the memory allocation function.
  - Systematically replace every access—on the right or the left hand side of any statement—to the variable S_i (i.e., an expression of the form $S_i[f(z)]$ for some affine function f) by $S_i[\mathcal{M}_i(f(z))]$.

9.4 Scanning a union of polyhedra

The key unresolved problem in the code generation algorithm outlined above is that of generating code to visit the points of a *union* of polyhedra, as required in the third step above. We now address this problem, building on the solution to the (single) polyhedron-scanning problem seen earlier.

We first note that the problem is not as easy as it seems at first glance. We cannot simply scan the individual polyhedra separately and some how "combine" the solution. For example, consider the domains $\mathcal{D}_1 = \{$`i| 1<=i<=10`$\}$ and $\mathcal{D}_2 = \{$`i| 4<=i<=12`$\}$, with respective schedules $\Lambda_1(i) = i + 1$ and $\Lambda_2(i) = 2i$. The statements associated with points 1 and 10 of $\mathcal{D}_1$ have to be executed at logical time 2 and 11, while the statements associated with points 4 and 8 of $\mathcal{D}_2$ have to be executed at times 8 and 16. Since $1 < 8 < 10 < 16$, we cannot first completely scan $\mathcal{D}_1$, and then scan $\mathcal{D}_2$, and neither can we scan $\mathcal{D}_2$ followed by $\mathcal{D}_1$. Hence, it is not possible to generate separate loops to scan both $\mathcal{D}_1$ and $\mathcal{D}_2$. These loops must be partially merged, so that the statements are executed in the order given by the schedule.

(a) Statements qualified by domains

```
{i,j | 1<=i<=N; 1<=j<=M} :: S1;
{i,j | i=j; 3<=j<=N} :: S2;
```

(b) View of the domains for the case where $4 \leq M \leq N$

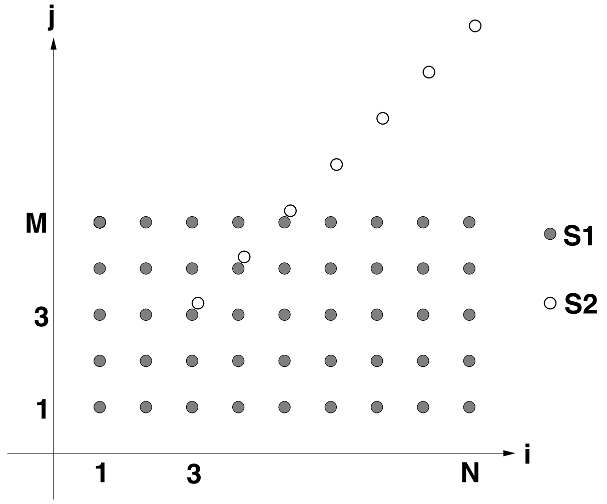


Figure 7: Systems of statements qualified by domains

The simplest solution to scanning a union of polyhedra uses a perfectly nested loop to scan a convex superset of this union. This superset may be either the bounding box of the domain (Fig. 8.a), or the convex closure of the domain (Fig. 8.b). However, since the domain scanned by this loop is also a superset of *each* statement's domain, we cannot unconditionally execute the statements within the loop body. Rather, each statement S_i must be guarded by conditions which test that the current loop index vector belongs to $\mathcal{D}_i$.

However, this solution yields inefficient (albeit compact) code because of the following limitations.

- Empty iterations: a perfectly nested loop scans a convex polyhedron. Since a domain that is a union of polyhedra may not be convex, some iterations of the loop may not execute any statements. It is often especially inefficient when a domain has an extreme aspect ratio. For example in Fig. 8.a, only about `M` out of every `N` points are usefully visited. Depending on the parameter values (say if `M` $\ll$ `N`) and on the complexity of the loop body, this could have a significant overhead.

- Control overhead: each loop iteration must test the guards of all guarded statements, causing a significant control overhead.

- Finally, finite unions of finite parameterized polyhedra do not always admit finite

(a) bounding box

```
for (i=1; i<=N; ++i){
  for (j=1; j<=N; ++j){
    if (j<=M)
      S1;
    if (i==j && i>=3)
      S2;
  }
}
```

(b) convex closure

```
for (i=1; i<=N; ++i) {
  for (j=1; j<=min ((i+3N+M-4)/4, i+M-1, N); ++j) {
    if (j<=M)
      S1;
    if (i==j && i>=3)
      S2;
  }
}
```

Figure 8: Scanning the union of domains of Fig. 7 by visiting points in a convex superset

convex supersets. Consider for example the following parameterized domains : {`i| 1<=i<=N`} and {`i| 1<=i<=M`}; the smallest convex super-set of these domains is the infinite polyhedron {`i| 1<=i`} (note that {`i| 1<=i<=max(M,N)`} is not a convex polyhedron).

To avoid this overhead, we can separate a union of polyhedra into several distinct regions that can each be scanned using imperfectly nested loops. The resulting code is more efficient, since:

- Imperfectly nested loops can scan non-convex regions, avoiding empty iterations;
- It may be possible to choose these loops such that some statement guards become always true or always false; when a guard is always true, the guard may be removed; when a guard is always false, the entire statement can be removed. This optimization increases the ratio of the number of executed statements to the number of tested guards, and thus reduces the control overhead. This can be carried through to its logical limit to yield code with no guards.

However, the efficiency is obtained at the expense of code size for two reasons. First, a perfect loop is replaced by an imperfectly nested loop, which scans a disjoint union of polyhedra. Second, a statement domain may be divided into several disjoint polyhedra, where each polyhedron is scanned by a different loop. In this case, the statement code has to be duplicated in each of these loops. Fig. 9 illustrates this for the example of Fig. 7.

```
for (i = 1; i <= 2; ++i) {
  for (j = 1; j <= M; ++j) {
    S1;
  }
}
for (i = 3; i <= N; ++i) {
  for (j = 1; j <= min((-1 + i),M); ++j) {
    S1;
  }
  if (i<=M) {
    j = i;
    S1;
    S2;
  }
  if (i>=M+1) {
    j = i;
    S2;
  }
  for (j = (1 + i); j <= M; ++j) {
    S1;
  }
}
```

Figure 9: Imperfectly nested loops scanning the union of domains of Fig. 7

The main idea is to recursively decompose the union of polyhedra into imperfectly nested loops, starting from outermost loops to innermost loops. At each step, we seek to solve the following problem: given a context (i.e., a polyhedron in $\mathbb{Z}^{(d-1)}$ containing the outer loops and system parameters), and a union of polyhedra in $\mathbb{Z}^n, n \geq d$, generate a loop that scans this union in lexicographic order.

We generate each additional level of loops by:

1. Projecting the polyhedra onto the outermost d dimensions;

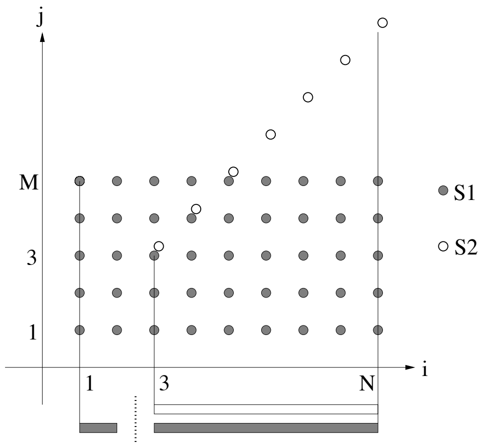
2. Separating these projections into disjoint polyhedra;

3. Recursively generating loop nests that scan each of these;

4. Sorting these loops so that their textual order respects the lexicographic order.

There are two subtle details that arise. First, the idea of "sorting" polyhedra (in step 4), necessitates an order relation and an associated algorithm. Second, we need to ensure that

(a) Projection on `i` and separation into disjoint polyhedra



(b) Sorted `i` loops

```
{i | 1<=i<=2} ::
  {i,j | 1<=i<=2; 1<=j<=M} ::
    S1;
{i | 3<=i<=N} ::
  {i,j | 3<=i<=N; 1<=j<=M} ::
    S1;
  {i,j | 3<=i<=N;  i=j} ::
    S2;
```

Figure 10: Projections of domains on first dimension and separation into disjoint loops

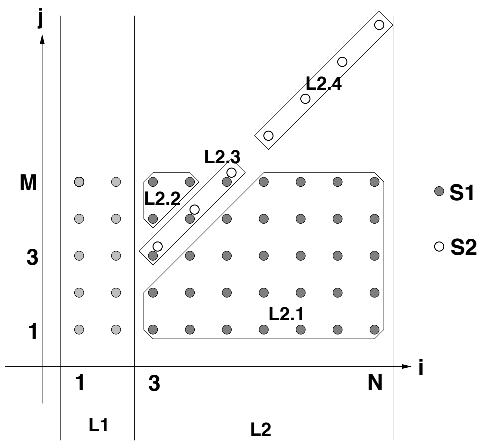
the "separation" of polyhedra (in step 2) must be such that the resulting polyhedra can be so sorted.

We illustrate the algorithm with an example. Let us start with the program of Fig. 7, and generate the first level of loops. Each domain is projected onto the first dimension and the parameters, i.e., {`i,N,M`}. These projections are then separated into three disjoint regions: one region containing only `S1`, one region containing both `S1` and `S2`, and one region containing only `S2`. For our example, the last region is empty, as shown in Fig. 10.a. Furthermore, note that some regions (those that involve the difference of polyhedra) could be unions of polyhedra.

The two remaining regions represent two loops scanning different pieces of dimension `i`, parameterized by `N` and `M`. At this point, only the first level polyhedra ({`i | 1<=i<=2`} and {`i | 3<=i<=N`}) can be interpreted as loops. The other (2-dimensional) polyhedra act as guards on the statements. Note that in Fig. 10, these guards are partially redundant with their context. The redundant constraints are eliminated later.

We will recursively generate separate pieces of code to scan, respectively, regions 1 (containing `S1` alone) and 2 (containing both `S1` and `S2`). Then we will, textually place the former *before* the latter. This order respects the lexicographic schedule. Finding such a textual order is trivial for the first dimension, but becomes more complicated for subsequent

(a) Disjoint polyhedra



(b) Sorted loops

```
L1    {i | 1<=i<=2} ::
L1.1    {i,j | 1<=j<=M} ::
          S1;
L2    {i | 3<=i<=N} ::
L2.1    {i,j | 1<=j<=i-1; j<=M} ::
          S1;
L2.3    {i,j | i=j; j<=M} ::
          S1;
          S2;
L2.4    {i,j | i=j; M+1<=j} ::
          S2;
L2.2    {i,j | i+1<=j<=M} ::
          S1;
```

Figure 11: Second level: projection and separation of the second loop nest (`L2`)

levels. In the rest of this example, whenever such an order is needed, we will give a valid order without discussing how to obtain it.

Now, we generate the next level of loops in the context of the first level loops. The first `i`-loop (labeled `L1` in Fig. 11(a)) contains only one statement, and thus the generation of its inner loop is a perfect loop generation problem. Next, consider the second i-loop, `L2`. It contains two guarded statements. In the next level of transformation, these domains are first separated into four disjoint polyhedra: two of them (namely `L2.1` and `L2.2`) containing `S1` alone, one (`L2.3`) containing both `S1` and `S2`, and one (`L2.1`) containing `S2` alone. Then, we sort these polyhedra, such that the following constraints are respected.

- Loop (`L2.1`) precedes loops (`L2.2`), (`L2.3`) and (`L2.4`).
- Loop (`L2.3`) precedes loop (`L2.2`).

There exist multiple valid textual orderings of these four loops, and we propose one of them in Fig. 11.(b). Also note that there is no constraint on the relative position of loops (`L2.2`) and (`L2.4`), since their respective contexts ({`i|i<=M-1`} and {`i|1>=M+1`}) are distinct. In other words, for any value of the parameter `M` and the outer loop index `i`, at least one of these loops is empty; it follows that any textual order of these loops will execute correctly.

Finally, we generate code by pretty printing the sorted nested loops, yielding the code we saw previously in Fig. 9.

10 Limitations of the polyhedral model

The polyhedral model suffer from two important limitations. The first is inherent in the model, namely the restrictive class of programs covered. This is indeed a fundamental limitation, although one may perform an *approximate* analysis by investigating cases where the dependences in the program as well as the domains of iterations are approximated respectively by by affine functions and polyhedra. Nevertheless, the model precludes computations that have a more dynamic behavior, in the sense that the control flow is conditioned by results computed during the program. In spite of this limitation, the very large number of programs that come within its scope is considerable. One might view the limitation as the price to pay for the powerful analysis techniques that the model offers.

The second, and somewhat more serious limitation is that neither the analyses nor the program transformations that the model offers can satisfactorily deal with resource constraints. We have already had an inkling of this when we considered the processor allocation function in Section 7, where we were able to formalize and reason about the potential conflicts and the communication behavior only for a set of *virtual* processors. One might question whether choosing an optimal mapping to these virtual processors will remain optimal after the virtual-to-physical mapping that is expected to follow. Moreover, since this virtual-to-physical mapping will assign the computations of many virtual processors to a single physical processor, it will also be necessary to modify the schedule. Hence our choice of the so called *optimal* schedules that we made in Section 6 is also open to question—the machine model there assumed an unbounded number of processors.

The most well known method for dealing with resource constraints is through a technique called tiling or blocking or super-node partitioning. Essentially, the idea is to "cluster" a certain number of nodes in the EDG as a single node and try to develop compact analysis methods. The common method of specifying tiles is as groups of computation nodes delineated by a family of hyperplanes (i.e., integer points within a hyper-parallelepiped shaped subset of the iteration space). Unfortunately such transformations do not satisfy the closure properties of the polyhedral model. This is especially true if we wish to retain both the size and the number of tiles in each dimension as a *parameter* for the analysis and hence develop compact methods for analysis and code generation.

11 Bibliographic notes

There are a number of references on the more conventional aspects of dependence analysis and restructuring compilers, notably the texts by Banerjee [6, 7] Zima and Chapman [49], and recently by Allen and Kennedy [3]

The roots of the polyhedral model can be traced to the seminal work of Karp Miller and Winograd [22], who defined SURE's, and studied, for SURE's where all variables are defined over the entire positive orthant, resolved the scheduling problem. They developed the multidimensional scheduling[9] algorithm explained in Section 6.

Another field that contributed heavily to the polyhedral model was the work on automatic synthesis of systolic arrays, which initially built on the simpler ideas (1-dimensional affine schedule for a single SURE, and shifted-linear schedules for SARE's) of Karp et. al. Though many authors in the early 80's worked on the systolic synthesis by space time transformations, the first use of recurrence equations was in 1983 by Quinton [33], who studied a single URE[10] defined over a polyhedral index space. He expressed the space of one-dimensional affine schedules as the feasible space of an integer linear programming problem, thus using establishing the link to the work of Karp et al. Rao [39, 40] and Roychowdhury [42] investigated multidimensional schedules for SURE's, and Rao also improved the alorithm given by Karp et al.

Delosme and Ipsen first studied the scheduling problems for affine recurrences (they defined the term ARE), but without considering the domains over which they were defined [14]. They showed that all (one-dimensional) affine schedules belong to a cone. Rajopadhye et al. addressed the problem of scheduling a single ARE defined over a polyhedral index domain and showed that the space of valid one dimensional affine schedules is described in

[9]Today we know them to be multidimensional schedules, but this understanding was slow in coming.

[10]Actually Quinton worked on a a particular restricted form of SURE's, but this can always be viewed as a single URE for analysis purposes.

terms of linear inequalities involving the extremal points (vertices and rays) of the domain [38]. Quinton and van Dongen also obtained a somewhat tighter result [34]. Yaacoby and Cappello [48] investigated a special class of ARE's. Mauras et al. extended this result to SARE's, but with 1-dimensional *variable dependent* schedules [30]. Rajopadhye et al. further extended this and proposed *piecewise* affine schedules for SARE's [37]. Feautrier [18] gave an alternative formulation using Farkas' lemma, for determining (one-dimensional, variable dependent) affine schedules for a SARE. He further extended the method to multidimensional schedules [19]. Some similar ideas were also developed in the loop parallelization community, notably the work of Lamport [24], Allen and Kennedy [1, 2] and Wolf and Lam [45]. An excellent recent book by Darte, Robert and Vivien [12] provides a detailed description of scheduling SURE's, and also addresses the problem of SARE's by formalizing affine and more general dependences as *dependence cones* engendered by a finite set of uniform dependence vectors.

Feautrier [16] paved the way to using these results for loop parallelization by developing an algorithm for exact data-flow analysis of ACL's (which he called "static control loops", somewhat of a misnomer) using parametric integer programming, and showed that this yields a SARE whose variables are defined over (a generalization of) polyhedral domains. The ideas in Section 4 are based on his work, though presented differently.

In terms of the decidability of the scheduling problem, Joinnault [21] showed the undecidability of SURE scheduling when the variables are defined over arbitrary domains. Quinton and Saouter [43] showed that scheduling even parameterized families of SARE's (each of whose domains is bounded) is also undecidable.

The processor allocation function as defined in Section 7 is directly drawn from the systolic synthesis literature. Modeling the communication as the image of the dependences by the allocation function also draws from work on the localization of affine dependences, notably by Fortes and Moldovan [20], Choffrut and Culik [10] Rajopadhye et al. [38, 35, 36], Quinton and Van Dongen [34], Wong and Delosme [46], Roychowdhury et al. [42, 41], and Yaacoby and Cappello [47], and Li and Chen [28, 27].

The problem of memory allocation in the polyhedral model has been addressed by Chamski [8], De Greef and Catthoor [13], Lefebvre and Feautrier [26], and Quillere, Rajopadhye and Wilde [44, 31] (from which the results presented in Section 8 are drawn).

The development of a programming language and transformation system based on the polyhedral model also draws considerably from the area of systolic synthesis. Some early results on the closure of a single ARE were developed by Rajopadhye et al. [38]. Chen defined the language Crystal based on recurrences defined over data fields [9], and Choo and Chen developed a theory of domain transformations for Crystal. ALPHAwas defined by Mauras [29] and developed an initial transformation framework for ALPHA programs.

The key problem of code generation, namely polyhedron scanning was posed and resolved by Ancourt and Irigoin [5] using the Fourier-Motzkin elimination. It is used in many parallelizing tools such as the SUIF system [4] developed at Stanford University, PIPS developed at the École des Mines de Paris, LOOPO at the University of Passau, etc. The efficient formulation using the dual representation was developed by LeVerge et al. [25]. Code generation from a union of polyhedra was addressed by Kelly et. al [23] for the Omega project at the University of Maryland. and by Quilleré et. al [32] (the results presented in Section 9 are drawn from the latter).

Additional information about the polyhedral model may be obtained from a survey article by Feautrier [17], and many of the mathematical foundations are described by Darte [11].

References

[1] R. Allen and K. Kennedy. PFC: A program to convert Fortran to parallel form. In *IBM Conference on Parallel Processing and Scientific Computing*, 1982.

[2] R. Allen and K. Kennedy. Automatic translation of Fortran programs to parallel form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[3] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann, 2002.

[4] S. P. Amarasinghe. *Parallelizing Compiler Techniques based on Linear Inequalities.* PhD thesis, Stanford University, Computer Science Department, 1997.

[5] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Third Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 39–50. ACM SIGPLAN, ACM Press, 1991.

[6] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[7] U. Banerjee. *Loop Transformations for Restructuring Compilers.* Kluwer Academic Publishers, 1993.

[8] Z. Chamski. Generating memory efficient imperative data structures from systolic programs. Technical Report PI-621, IRISA, Rennes, France, December 1991.

[9] Marina C. Chen. A parallel language and its compilation to multiprocessor machines for VLSI. In *Principles of Programming Languages.* ACM, 1986.

[10] C. Choffrut and K. Culik II. Folding of the plane and the design of systolic arrays. *Information Processing Letters*, 17:149–153, 1983.

[11] A. Darte. Mathematical tools for loop transformations: from systems of uniform recurrence equations to the polytope model. In M. H. Heath, A. Ranade, and R. Schreiber, editors, *Algorithms for Parallel Processing*, volume 105, pages 147–183. IMA Volumes in Mathematics and its Applications, 1999.

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

[13] Eddy De Greef, Francky Catthoor, and Hugo De Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. In *Parallel Processing and Multimedia*, Geneva, Switzerland, July 1997.

[14] Jean-Marc Delosme and Ilse C. F. Ipsen. Systolic array synthesis: Computability and time cones. In M. Cosnard, P. Quinton, Y. Robert, and M. Tchuente, editors, *Int. Workshop on Parallel Algorithms and Architectures*, pages 295–312. Elsevier Science, North Holland, April 1986.

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

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

[17] P. Feautrier. Automatic parallelization in the polytope model. In G-R. Perin and A. Darte, editors, *The Data Parallel Programming Model: CNRS Spring School*, pages 79–103. Springer Verlag, Les Ménuires, France, March 1996.

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

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

[20] Jose A. B. Fortes and D. Moldovan. Data broadcasting in linearly scheduled array processors. In *Proceedings, 11th Annual Symposium on Computer Architecture*, pages 224–231, 1984.

[21] P. Gachet and B. Joinnault. *Conception d'algorithmes et d'architectures systoliques.* PhD thesis, Université de Rennes I, September 1987. (In French).

[22] R. M. Karp, R. E. Miller, and S. V. Winograd. The organization of computations for uniform recurrence equations. *JACM*, 14(3):563–590, July 1967.

[23] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers 95: 5th Symposium on the Frontiers of Massivelly Parallel Computation*, McLean, VA, 1995.

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

[25] H. Le Verge, V. Van Dongen, and D. Wilde. Loop nest synthesis using the polyhedral library. Technical Report PI 830, IRISA, Rennes, France, May 1994. Also published as INRIA Research Report 2288.

[26] Vincent Lefebvre and Paul 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.

[27] J. Li and M. Chen. The data alignent phase in compiling programs for distributed memory machines. *Journal Parallel Distributed Computing*, 13:213–221, 1991.

[28] J. Li and M. C. Chen. Compiling communication efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[29] Christophe Mauras. *ALPHA: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, L'Université de Rennes I, IRISA, Campus de Beaulieu, Rennes, France, December 1989.

[30] Christophe Mauras, Patrice Quinton, Sanjay V. Rajopadhye, and Yannick Saouter. Scheduling affine parameterized recurrences by means of variable dependent timing functions. In S. Y. Kung and E. Swartzlander, editors, *International Conference on Application Specific Array Processing*, pages 100–110, Princeton, New Jersey, Sept 1990. IEEE Computer Society.

[31] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems*, 22(5):773–815, September 2000.

[32] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, October 2000.

[33] P. Quinton. The systematic design of systolic arrays. In F. Fogelman Soulie, Y. Robert, and M. Tchuente, editors, *Automata Networks in Computer Science*, chapter 9, pages 229–260. Princeton University Press, 1987. Preliminary versions appear as IRISA Tech Reports 193 and 216, 1983, and in the proceedings of the IEEE Symposium on Computer Architecture, 1984.

[34] 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.

[35] S. V. Rajopadhye. Synthesizing systolic arrays with control signals from recurrence equations. *Distributed Computing*, 3:88–105, May 1989.

[36] S. V. Rajopadhye. LACS: A language for affine communication structures. Technical Report 712, IRISA, 35042, Rennes Cedex, April 1993.

[37] S. V. Rajopadhye, L. Mui, and S. Kiaei. Piecewise linear schedules for recurrence equations. In K. Yao, R. Jain, W. Przytula, and J. Rabbaey, editors, *VLSI Signal Processing, V*, pages 375–384, Napa, Oct 1992. IEEE Signal Processing Society.

[38] 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.

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

[40] Sailesh Rao and Thomas Kailath. What is a systolic algorithm. In *Proceedings, Highly Parallel Signal Processing Architectures*, pages 34–48, Los Angeles, Ca, Jan 1986. SPIE.

[41] Vwani Roychowdhury, Lothar Thiele, Sailesh K Rao, and Thomas Kailath. On the localization of algorithms for VLSI processor arrays. In Robert W. Brodersen and Howard S. Moscovitz, editors, *VLSI Signal Processing, III*, pages 459–470, Monterey, Ca, November 1988. IEEE Accoustics, Speech and Signal Processing Society, IEEE Press. A detailed version is submitted to IEEE Transactions on Computers.

[42] Vwani P. Roychowdhury. *Derivation, Extensions and Parallel Implementation of Regular Iterative Algorithms*. PhD thesis, Stanford University, Department of Electrical Engineering, Stanford, CA, December 1988.

[43] Y. Saouter and P. Quinton. Computability of recurrence equations. *Theoretical Computer Science*, 114, 1993.

[44] D. Wilde and S. Rajopadhye. Memory reuse analysis in the polyhedral model. *Parallel Processing Letters*, 7(2):203–215, June 1997.

[45] M. Wolf and M. Lam. Loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, Oct 1991.

[46] F. C. Wong and Jean-Marc Delosme. Broadcast removal in systolic algorithms. In *International Conference on Systolic Arrays*, pages 403–412, San Diego, CA, May 1988.

[47] Yoav Yaacoby and Peter R. Cappello. Converting affine recurrence equations to quasi-uniform recurrence equations. In *AWOC 1988: Third International Workshop on Parallel Computation and VLSI Theory*. Springer Verlag, June 1988. See also, UCSB Technical Report TRCS87-18, February 1988.

[48] Yoav Yaacoby and Peter R. Cappello. Scheduling a system of nonsingular affine recurrence equations onto a processor array. *Journal of VLSI Signal Processing*, 1(2):115–125, 1989.

[49] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, Frontier Series, 1990.