Transforming Affine Nested Loop Programs to Dataflow Computation Model

Arkady Klimov

Institute of Design Problems in Microelectronics,
Russian Academy of Sciences, Moscow, Russia
arkady.klimov@gmail.com

Abstract. Programs in dataflow computation model are easy to parallelize. Hence the problem of parallelizing a sequential program can be reduced to the problem of transforming it to dataflow computation model. As target model we use a computation model of the Parallel Dataflow Computing System (PDCS) developed at IDPM RAN. In this paper a method of mapping a sequential Fortran program to the target dataflow language is described. The source programs must form a set of nested loops with the loop bounds and array indices being affine functions of loop indices.

Keywords: parallelizing affine loops, polyhedral model, dataflow architecture.

1 Introduction

The well-known problem of automated parallelization of sequential programs [1] has not received a satisfactory solution yet, especially in the domain of multiprocessors with distributed memory. What makes it so difficult? I believe the main complication is the target parallel programming model. It usually requires the programmer's full control over the execution scheduling. Consider, for example, the MPI model, which has been widely used for the last decades, and the GPGPU model developed recently. While demonstrating more data parallelism and efficiency, the latter model, as compared with the former one, forces the programmer to control more details of the space-time layout of computation and communication. The trend is towards more thorough control over more and more aspects. Accordingly, the problem of automatic parallelization [2] becomes more complicated.

The matter with dataflow model is quite different. The programming in it is considered difficult due to its unaccustomedness rather than to its intrinsic features. And the layout details here are hidden for the programmer. Though making some of them visible may be useful, a lot of these details are still out of the static (programmer or compiler) control. Accordingly, almost all of these details are to be controlled by runtime system, foremost by hardware.

The main programmer concern in dataflow model is about the links, dependencies between data objects (to say nothing about computations proper). In our model the link is defined in the context of producer. In other words, each producer must know where its results will be used. However, in traditional sequential languages, like Fortran, an opposite paradigm is used: normally it is the consumer who initiates

access to data which is stored somewhere by a producer. We shall call the latter paradigm *gathering*, as opposed to *scattering* paradigm which is the basis of our dataflow model. From now on we shall use acronym DFL (a DataFlow Language) as the name of a programming language for our computation model.

To transform a sequential Fortran program to DFL one must both extract its inherent dataflow graph and invert the direction of the graph. And first of all we need a means to represent the graph. In the paper we describe a method of building the graph of sequential program and then show its use for generating equivalent DFL program. In section 2 we define briefly the dataflow computation model and the language DFL. Then, in section 3 we restrict the domain of all Fortran programs to the so-called *affine programs* which are both usable in practice and amenable to automatic translation to DFL. Further sections uncover the process of the transformation. In section 4 the notion of *selection tree* is introduced and some operations over them are defined. The idea of selection tree is motivated by the notion of *semantic effect* of program statement, for which the selection tree is used as a representation. In section 5 the notion of effect is defined thoroughly together with a method to build corresponding selection tree (effect tree) for each program statement from bottom to top along its AST. Then in section 6 the effect trees are used to build the so-called *state* trees, which describe the state of memory at each program point. Using these it is easy to obtain, for each read operator in the program, a so-called LWT (Last-Write Tree), or a *source tree* (described in section 7) which altogether comprise the description of the data flow graph of the program. In fact it is the destination-to-source graph. But we need an opposite, the source-to-destination graph. And thus in section 8 we propose an extension of selection trees to multi-valued trees (former effect/state/source trees were single-valued) and use it to define the operation of graph inversion. Its result is a set of multi-valued trees, which can then be treated as dataflow program. In section 9 a straightforward process of transforming it into DFL is described. Section 10 contains an example of program dataflow analysis and transformation with the resulting parallelization effect. Further sections contain some discussions, sketch improvement directions and compare our approach to automatic parallelization with other existing ones.

2 Dataflow Computation Model

To present our computation model we use a programming language called DFL. A program in DFL is a set of *node declarations* of the form:

node *name ports context-dcl*;
routine;

The first line is a node *header*. It specifies node name, list of ports (port names and types) and context declaration. Context is a tuple of values (normally integers) which serve as indices of node. In fact, the node declaration defines a set (not necessarily finite) of node instances, or *virtual nodes*, that differ from each other with context. Context declaration in the header is just a list of context field names. Optionally the context type may be specified.

The node declaration can be viewed as a class declaration. Instances occur on demand when token is send to a port of a node with some context. In a sense, the context provides a virtual address of node instance. The ensemble of all possible instances of all nodes declared in a program comprises the total virtual node space of the program.

Node ports are like procedure parameters with the only difference that any port of any virtual node is invoked independently. Such invocations will be referred to as *tokens*. The node instance can *fire* when each of its ports gets a token. All these tokens must have common context values and the same node class name. At a moment of time there may be several virtual nodes ready to fire. They may fire in any order or in parallel.

When node instance fires, a request is created which is then ready to be executed. At the same time all tokens involved in the firing are removed from the virtual node ports, while their values are copied into request. (If there occur several tokens on some port of given node instance at the moment of firing, then arbitrary one of them is used in the firing and others remain waiting for further firings). At a moment of time there may be any number of ready requests, which may be then executed in any order or in parallel. To execute the request means to execute the node's *routine*. Routine is a normal sequential code which operates locally using port names and context field names as input variables. The only "side effect" of request execution may be a number of tokens sent to other virtual nodes' ports. Thus all requests are executed independently.

The token send statement has the form:

$$v \texttt{ -> } N\texttt{.}p\texttt{\{}e_1\texttt{,}\ldots\texttt{,}e_k\texttt{\}}$$

where the type of expression v is the same as that of port p of node N and integer expressions $e_1,\ldots,e_k$ yield the context of the target virtual node. It reads: "send v to port p of node N with context $\{e_1,\ldots,e_k\}$". Or shorter: "send v to $N.p\{e_1,\ldots,e_k\}$". The token send statement is not blocking. The token created moves by itself to indicated virtual node port. No assumptions are made on behalf of the time needed by a token to get to its target port and of the order different tokens arrive. At any moment of time there is a number of moving tokens and a number of tokens residing at virtual node ports.

The total computation is initiated by sending a number of input tokens from outside and results in creating a number of output tokens. For certainty we assume that input tokens are sent to special 1-port nodes whose name has suffix "_in" and output tokens are sent to special 1-port nodes which have empty routine and name with suffix "_out". Normal *good* program should terminate with no tokens residing anywhere in the virtual space.

A program may also contain some constant declarations with keyword **`vconst`** (see example below), which may vary from session to session. Their exact values are specified from outside the system before the session.

In Fig.1 an example program in DFL is shown. There are 3 nodes including one input and one output nodes. The program adds n numbers which come from outside as

$$a_i \texttt{ -> X1\_in\{}i\texttt{\}}$$

for i=0,…n-1 (port name for 1-port node may be omitted). The resulting sum is sent back via node `Sum_out` (with empty context). Pascal-like syntax is used for

encoding routine which is a single statement or block preceded by some local declarations. This program is highly parallel: if each virtual node executes in its own processor the overall time would be $O(\log n)$ after last input token arrive. Doubling method is used. Below we'll show a sequential Fortran program which this example can be compiled from.

```
vconst n:int;
node X1_in(b:real){i};
  if i mod 2 = 0
  then b -> X2.b{i div 2}
  else b -> X2.a{(i+1) div 2};
node X2(a:real,b:real){i};
  if i=n-1
  then a+b -> Sum_out {}
  else if (i+n) mod 2 = 0
  then a+b -> X2.b{(i+n) div 2}
  else a+b -> X2.a{(i+n+1) div 2};
node Sum_out(s:real){};
```

Fig.1. An example: parallel summation in DFL

3 Affine Programs

In DFL a node routine must be able to send computed value to any other node that needs it. Hence to transform a Fortran program P to DFL one needs to predict, for each *write* operation (i.e. assignment statement), the set of all *read* operations that will use the value written. In fact, a description of *algorithm graph* ([3, chapter 6]) must be built, which parametrically describes the computation graph. The *computation graph* is defined by simply running the program P for some input data. It consists of two kinds of nodes: reads and writes, corresponding respectively to executions of read or write memory operations in program P. There is a link from a write node w to a read node if r reads the value written by w. In other words, r uses the same memory cell as w and w is the last write to this cell before r.

We identify a separate read or write operation by a pair (m,I_m), where m is a program point and I_m is an iteration vector of point m. (*Iteration vector* of point m is a list of loop index values for all loops enclosing point m.) Using this pairs as graph node identifiers we define the mapping

$$\mathbf{F}_P : (r,I_r) \rightarrow (w,I_w) \tag{1}$$

which for a read node (r,I_r) yields the write node (w,I_w) that has written the value being read, or yields **None** if no one value has been written before the read and thus original contents of the cell is read.

However, for translation to DFL we need the reverse: for each write node to find all read nodes (and there may exist several or none of them) which read that very value written. So, we need the multi-valued mapping

$$\mathbf{G}_P : (w,I_w) \rightarrow \{(r,I_r)\} \tag{2}$$

which for each write node (w,I_w) yields a set of all read nodes $\{(r,I_r)\}$ that read that very value written.

Unfortunately, it is hardly possible for an arbitrary program P to express these mappings $\mathbf{F}_P$ and $\mathbf{G}_P$ explicitly and finitely. But there exist a well known and well defined class of programs for which it is possible: the so-called *affine* (or *linear* as in [3, chapter 6]) programs. The set of constructors used to build a statement S in our subset of Fortran is presented on Fig.2.

Λ	(empty statement)
$A(i_1,\ldots,i_k)=e$	(assignment, $k\geq 0$)
$S_1;\ S_2$	(sequence)
if c then S_1; else S_2; endif	(conditional)
do $v=e_1,e_2$; S; enddo	(DO-loop)

Fig.2. Affine program constructors.

The right hand side e of assignment may contain array element access $A(i_1,\ldots,i_k)$, $k\geq 0$. All index expressions must be affine (see below). Conditional expression c must be equivalent to $e=0$ or $e>0$ where e is an affine expression. Bounds e_1 and e_2 of DO-loop must also be affine. Expression is *affine* if it is a sum of enclosing loop variables or fixed parameters with literal integer coefficients.

At present, our framework admits only purely affine subroutines with all input and output arrays passes as parameters. The transformer replaces the subroutine with another one of the same name which passes all input array elements as tokens to parallel subsystem and accepts results. The respective DFL code is generated and loaded into the subsystem. In the future some of the restrictions may be lifted. In particular, the analyzer will find the code region to be translated to DFL itself.

4 Selection Trees and Operations on Them

A selection tree [4] is a structure used to represent mappings like (1). Its syntax is shown in Fig.3.

S-tree ::= **None**	
| *term*	
| (*L-cond* → *S-tree*$_t$: *S-tree*$_f$)	(branching)
| (*L-expr* =: *num var* + *var* → *S-tree*)	(integer division)
term ::= *name* { *L-expr*$_1$, … , *L-expr*$_k$ }	($k\geq 0$)
var ::= *name*	
num ::= … | -2 | -1 | 0 | 1 | 2 | 3 | …	
L-cond ::= *L-expr* = 0 | *L-expr* > 0	(affine condition)
L-expr ::= *num* | *num var* + *L-expr*	(affine expression)
atom ::= **None** | *name* { *num*$_1$, … , *num*$_k$ }	(ground term, $k\geq 0$)

Fig.3. Syntax of Selection Trees

A branching node like ($c \rightarrow T_1 : T_2$) evaluates to T_1 if conditional expression c evaluates to true, otherwise to T_2.

A division node like ($e =: 2\ \mathbf{q} + \mathbf{r} \rightarrow T$) introduces two new variables **q**,**r** that take respectively the quotient and the remainder of integer division of integer value of e by constant 2. They may be used in the sub-tree T together with all other variables the tree depends on. But the whole tree does not depend on variables **q** and **r** as they are *bound variables*. Notes on font styles: we normally use small bold letters as variables of our object language while italic letters stand for variables in the meta-language denoting an arbitrary object of certain kind. Hence bold italic denotes 'arbitrary object variable' and various capitals represent structures: vectors, lists, trees etc. Other symbols or words usually stand for specific constants, function names etc.

Evaluation of a selection tree T may be specified by function $\mathcal{F}[T,\boldsymbol{V}]$ that maps a tuple of integer values J (one value for each variable in $\boldsymbol{V}$) to the domain of atoms. The variable list $\boldsymbol{V}$ must contain all free variable of tree T. We call the occurrence of variable $\boldsymbol{v}$ free if it is not bound outside by a division node. To apply function $\mathcal{F}[T,\boldsymbol{V}]$ to number vector J we assign values J_i to variables $\boldsymbol{V}_i$, evaluate all expressions to numbers and then simplify the result to atom. The formal definition of symbol $\mathcal{F}$ is presented in Fig.4.

$\mathcal{F}[\ \mathbf{None},\boldsymbol{V}\ ]J = \mathbf{None}$

$\mathcal{F}[\ \mathrm{X}\{e_1, \ldots, e_k\},\boldsymbol{V}\ ]J = \mathrm{X}\{z_1,\ldots, z_k\},$ where $z_i = \mathcal{F}[e_i,\boldsymbol{V}\ ]J$

$\mathcal{F}[(c \rightarrow A{:}B),\boldsymbol{V}\ ]J =$ if $\mathcal{F}[c,\boldsymbol{V}\ ]J$ then $\mathcal{F}[A,\boldsymbol{V}\ ]J$ else $\mathcal{F}[B,\boldsymbol{V}\ ]J$

$\mathcal{F}[(e =: m*\boldsymbol{q}+\boldsymbol{r} \rightarrow A),\boldsymbol{V}]J = \mathcal{F}[A,\boldsymbol{V}{+}{+}\boldsymbol{q}{+}{+}\boldsymbol{r}\ ]\ (J{+}{+}j_q{+}{+}j_r),$

where j_q, j_r are integer numbers s.t. $\mathcal{F}[e,\boldsymbol{V}\ ]J = m * j_q + j_r,\ 0 \le j_r < m$

"++" – append operator

Fig.4. Definition of selection tree semantic function $\mathcal{F}[T,\boldsymbol{V}]$. It is assumed that semantic functions of expressions $\mathcal{F}[E_i,\boldsymbol{V}\ ]J$ and conditions $\mathcal{F}[C_i,\boldsymbol{V}\ ]J$ are defined in a usual way.

We call two selection trees *equivalent* (≈) if their semantic functions are the same. Having this equivalence in mind we define several operations over selection trees. All operations must be defined correctly in the sense that if we replace an argument tree by equivalent one then the result should be equivalent to the former.

The two following operations play the main role in our framework: $seq(T_1,T_2)$ and $fold(\boldsymbol{v},e_1,e_2,T)$.

Operation $seq(T_1,T_2)$ unifies values of its arguments in such a way that the non-**None** value of T_2 has the priority over that of T_1. Formally:

$\mathcal{F}[seq(T_1,T_2),\boldsymbol{V}]J =$ if $\mathcal{F}[T_2,\boldsymbol{V}]J = \mathbf{None}$ then $\mathcal{F}[T_1,\boldsymbol{V}]J$ else $\mathcal{F}[T_2,\boldsymbol{V}]J$

To compute *seq* one may simply replace all **None** leaves in T_2 with a copy of T_1. As we shall see later *seq* corresponds to sequential execution of two statements. Note that *seq* is associative and has a unit: **None**. Thus it can be generalized to any number of argument trees as well as to list of trees.

Operation *fold*(***v***,e_1,e_2,*T*) besides a single tree argument *T* has three additional parameters: a variable ***v*** and two affine expressions e_1 and e_2 that do not depend on ***v*** while the tree *T* itself *may* depend on ***v***. The result is a tree *T'* that does not depend on ***v***. Its value must be as follows. Using assignment ***V***=*J* for all free variables of e_1, e_2 and *T* but ***v***, evaluate e_1 to n_1, e_2 to n_2 and consider sequence of values of *T*(***v***) for all ***v*** from n_1 to n_2 in that order. Take the last non-**None** value in the sequence if any, otherwise **None**.

The formal specification is

$$\mathcal{F}[\ fold(\boldsymbol{v},e_1,e_2,T),\boldsymbol{V}]J = seq(\{\ \mathcal{F}[\![T,\ \boldsymbol{V}{+}{+}\boldsymbol{v}]\!]J{+}{+}j \mid j \text{ in } (n_1..n_2)\}),$$

where $n_1 = \mathcal{F}[e_1,\boldsymbol{V}]J$, $n_2 = \mathcal{F}[e_2,\boldsymbol{V}]J$ and ++ is append operator. Here we do not require ***V*** not to contain ***v***. But we assume that the latest occurrence of a variable has priority in assignment ***V***=*J*. Thus, e_1 and e_2 *may* contain ***v***, but from an outer scope.

The computation of *fold* is not as simple. But the fundamental fact is that its result does always exist and can be computed by an algorithm. In theory, this fact stems from resolvability of Presburger arithmetics[5]. In practice, it resolves into Parametric Integer Programming (PIP) problem which is known to be solvable [6]. Below we shall see that *fold* is a counterpart to DO-loop statement.

There are also several helper operations on trees among which mention *equiv*(T_1,T_2) and *prune*(*T*).

Operation *equiv*(T_1,T_2) yields a so-called *predicate tree* that contains no terms but T{} and F{} only. It is specified formally as

$$\mathcal{F}[equiv(T_1,T_2),\boldsymbol{V}]J = \text{if } \mathcal{F}[T_1,\boldsymbol{V}]J = \mathcal{F}[T_1,\boldsymbol{V}]J \text{ then T}\{\} \text{ else F}\{\}$$

It is easy to see that $equiv(T_1,T_2) \approx \text{T}\{\}$ iff $T_1 \approx T_2$.

Operation *prune*(*T*) replaces a tree *T* by equivalent yet simpler one. It finds so-called imperfect conditional nodes (subtrees) ($c \rightarrow T_1 : T_2$) in which predicate *c* always evaluates to the same boolean value. Such node can be replaced by respective subtree (T_1 or T_2). The algorithm of *prune* traverses the tree and tests the set of conditions on each path for compatibility. The test that checks a set of affine equalities and inequalities for compatibility is known as *Omega test* [7]. It is a good idea to apply *prune* immediately after *seq*, *fold*, *equiv* and other complex operations.

5 Effect Trees

Consider a program statement `S`, which is a part of an affine program `P`, and some *k*-dimensional array `A`. Let (w_A, I_{wA}) denote an arbitrary write operation on an element of `A` within a certain execution of statement `S`, or the totality of all such operations. The effect of `S` with respect to `A` is a function

$$\mathcal{E}[S]: (p_1,\ldots,p_s;\ q_1,\ldots,q_k) \rightarrow (w_A,I_{wA}) + \textbf{None}$$

that for each tuple of fixed external parameters $p_1,\ldots,p_s$ and indices $q_1,\ldots,q_k$ of element of array `A` yields an element (w_A,I_{wA}) or **None**. The first result (w_A,I_{wA}) indicates that this very write operation is the last among those that write to element

A(q_1,...,q_k) during execution of S with parameters p_1,...,p_s and the result **None** means that there are no such operations.

To represent such functions it is feasible to use selection trees with program statement labels as term names. We shall call them simply effect trees. For example, consider as statement S the loop nest shown on Fig.5.

```
        do j=0,m-1
          do i=1,n
A1:         A(i+2*j) = some expression
          enddo
        enddo
```

Fig.5. An example loop nest S. The unique write statement is labeled as A1.

The set of all writes to array A (of size n+2*m) can be seen as that shown by Fig.6, where time runs from left to right and from top to bottom, and each colored square denotes a write to element A(q) depicted strictly above as a white square.

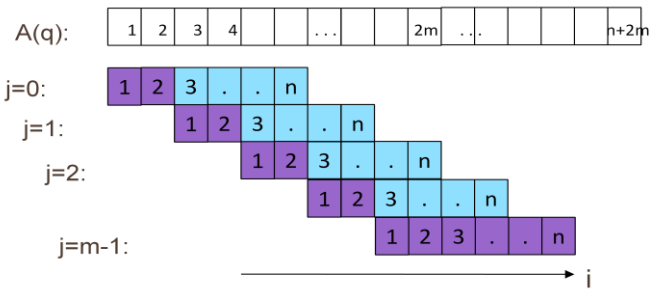

Fig.6. The totality of 'write to A' operations in the loop nest S from Fig.5.

The violet (darker) squares correspond to last writes to each element. It is easy to see that the respective effect may be represented as selection tree shown on Fig.7.

```
(q≥1 →
        (q≤2*m →
            (q-1=:2*j+r → A1{j,r+1})
            (q<=n+2*m-2 → A1{m-1,q-2*m+2} : None)
        )
: None)
```

Fig.7. Selection tree representing effect function of loop nest S from Fig.5.

Now that we have defined operations *seq* and *fold* the algorithm of computing effect tree of an arbitrary affine program fragment is straightforward. The set of rules listed in Fig.7 does the job recursively. Here we assume that effect tree is built for k-dimentional array A, k≥0. The result tree represents the statement' effect for element A(**q1**,…,**q**k), where **q1**,…,**q**k are fixed variables that are not used in the program.

The algorithm proceeds upwards from primitives like empty and assignment statements. Operation *seq* is used to compute effect of statement sequence having computed the effect of component statements. Operation *fold* yields the effect of DO-loop provided the effect of loop body. For conditional statement the effect is built just by putting effects of branches into new conditional node provided that condition c is affine.

$\mathcal{E}$[ ] = **None** (empty statement)

$\mathcal{E}[S_1; S_2] = seq(\mathcal{E}[S_1], \mathcal{E}[S_2])$ (sequence)

$\mathcal{E}[LA: A(e_1,\ldots,e_k)=e] =$ (assignments to A)

(**q1**=$e_1 \rightarrow \ldots$(**q**k=$e_k \rightarrow LA\{I\}$: **None**)…: **None**)

where I is a list of all outer loop variables of this statement in P

$\mathcal{E}[LB: B(\ldots)=e]$ = **None** (other assignments)

$\mathcal{E}$[if c then S_1; else S_2; endif] = ($c \rightarrow \mathcal{E}[S_1] : \mathcal{E}[S_2]$) (conditional)

$\mathcal{E}$[do $\boldsymbol{v}$=e_1,e_2; S; enddo] = $fold(\boldsymbol{v},e_1,e_2, \mathcal{E}[S])$ (DO-loop)

Fig.8. The rules for computing effect tree with respect to k-dimensional array A

6 State Trees

Consider an affine program P as a whole and a certain execution of it. Let there be a read operation of array element A($e_1,\ldots,e_k$) at point (r,I_r). We need to determine specific write operation (w,I_w) that has written the value being read.

We could achieve our goal if we knew this for each element of A. Thus we need to compute, for any given values of fixed parameters $\boldsymbol{P}=\boldsymbol{p}_1,\ldots,\boldsymbol{p}_s$ and array indices $\boldsymbol{Q}=\boldsymbol{q}_1,\ldots,\boldsymbol{q}_k$ and the point loop variables $\boldsymbol{V_r}=\boldsymbol{v}_1,\ldots,\boldsymbol{v}_l$, the coordinate ($w$,$I_w$) of write statement that has written last time on array element A($\boldsymbol{q}_1,\ldots,\boldsymbol{q}_s$) before the point ($r$,$I_r$). This information presents an effect of executing the program from the beginning up to the point (r,I_r) with respect to array A. It can be expressed as a selection tree with parameters $\boldsymbol{P},\boldsymbol{Q},\boldsymbol{V}$, which may be called a *state tree* at program point r for array A.

To compute state trees for each program point the following method can be used.

So far for each block statement B in the affine program P we have computed the tree $\mathcal{E}[B]$ representing the effect of B. Now we are to compute for each block statement B the tree $\mathcal{S}[B]$ representing the *state before* B.

We start by setting

$$\mathcal{S}[P] = (\boldsymbol{q_1} \geq l_1 \rightarrow (\boldsymbol{q_1} \leq u_1 \rightarrow \ldots (\boldsymbol{q_k} \geq l_k \rightarrow (\boldsymbol{q_k} \leq u_k \rightarrow \text{A\_init}\{\boldsymbol{q_1},\ldots,\boldsymbol{q_k}\} : \textbf{None}) : \textbf{None}) \ldots : \textbf{None}) : \textbf{None}) \quad (3)$$

where term A_init{$\boldsymbol{q_1},\ldots\boldsymbol{q_k}$} signifies an untouched value of array element A($q_1,\ldots,q_k$) and l_i, u_i are lower and upper bounds of array dimensions (which are only allowed to be affine functions of fixed parameters). This record signifies simply that all A's

elements are untouched before the whole program *P*.

Now consider the following cases:

1. Given $\mathcal{S}[B_1; B_2]=T$. Then also $\mathcal{S}[B_1]=T$. *The state before any starting part of* `B` *is the same as that before* `B`.
2. Given $\mathcal{S}[B_1; B_2]=T$. Then $\mathcal{S}[B_2]=seq(T, \mathcal{E}[B_1])$. *The state after the statement* B_1 *is that before* B_1 *combined by seq with the effect of* B_1.
3. Given $\mathcal{S}$[if *c* then B_1 else B_2 endif] = *T*. Then $\mathcal{S}[B_1] = \mathcal{S}[B_2] = T$. *The state before any branch of if-statement is the same as before the whole if-statement*.
4. Given $\mathcal{S}$[do $\boldsymbol{v}=e_1,e_2$; `B`; enddo] = T. Then $\mathcal{S}[B] = seq(T, fold(\boldsymbol{v},e_1,\boldsymbol{v}\text{-}1,\mathcal{E}[B]))$. *The state before the loop body B with the current value of loop variable* $\boldsymbol{v}$ *is that before the loop combined by seq with the effect of all preceding iterations of B*.

The last form needs some comments. It is the case in which a limit of *fold* depends on $\boldsymbol{v}$. Thus the result tree will also depend on $\boldsymbol{v}$. That tree involves the effect of all iterations of the loop before the current one.

Using the rules 1-4 one can achieve (and compute the state in) any internal point of program *P*. For speed, we do not compute the state with respect to array X in some point if there are no accesses to X within the current block after that point. Also, we compute only once the result of *fold* with variable upper limit and then use the result both for state at the beginning of the body and for the effect of the whole loop.

7 Destination-to-Source Graph

Now we are ready to compute the *destination-to-source* graph $\mathbf{F}_P$ of given affine program P, which is specified by formula (1). The representation of the graph is a collection of selection trees, one for each read operation *r* on array `X`. Each such tree T_r depends on fixed parameters $\boldsymbol{P}$ and loop variables $\boldsymbol{V}_r$ of loops enclosing *r*. A tree value indicates the write operation (w,I_w) that has written the value being read.

Note. One ought not to confuse our term *destination-to-sorce graph* with the term *def-use graph* of an arbitrary unstructured program, which is often used in papers on program analysis in optimizing compilers. In the latter case, nodes correspond to static program points, not to their dynamic execution instances as in our case.

To compute the tree T_r for a given read operation `X(`e_1`,…,`e_k`)` we simply take the state tree at immediately preceding program point and apply substitution $[\boldsymbol{q}_1 \rightarrow e_1, \ldots, \boldsymbol{q}_k \rightarrow e_k]$. After simplification with *prune* resulting trees are gathered as collection of elements of the form

$$((r,\boldsymbol{V}_r)\ C_r\ T_r), \tag{4}$$

where $C_r = (c_1,\ldots,c_K)$ is a list of restrictions that describe iteration space (a set of possible instantiations) of operation *r*. Usually this list contains bound conditions of enclosing loops and conditionals of enclosing ifs. Due to this restriction list the tree T_r itself can be freed of these conditions. This is done by the use of *conditional-prune* operation that eliminates spare conditions under given restrictions. As result, the tree T_r should *not* contain **None**. This is because the term **None** signifies, due to (3), that *r*

tries to read outside the range of array dimensions. Thus we get for free an easy way of compile-time verification of array access bounds (for reading).

For the graph to be complete we add a pseudo-read operation for any array element of any array X that was ever written. It is named (`X_out`, $(\boldsymbol{q_1},\ldots,\boldsymbol{q_k})$) and the restrictions involve the ranges of `X`'s dimensions. The tree is computed from the state tree at the very last point of program P, which is actually obtained as $seq(\mathcal{S}[\mathtt{P}],\mathcal{E}[\mathtt{P}])$. An additional analysis can be used to exclude arrays that are not used elsewhere. In our current implementation we restrict *out* arrays to subroutine parameter names.

8 Multi-valued Trees and Graph Inversion

Now we are to compute the *source-to-destination graph* $\mathbf{G}_\mathrm{P}$ of given affine program P, which is specified by formula (2). Conceptually, it is the inversion of destination-to-source-graph $\mathbf{F}_\mathrm{P}$. For representing it, however, we need another kind of selection trees which represent functions whose values are sets of atoms rather than just atoms. So, we extend the definition of selection (Fig.3) to multi-valued selection trees (Fig.9). Two new constructors are added: (& A_1 … A_n) collects together all values of subtrees $A_1,\ldots,A_\mathrm{n}$, and (@$\boldsymbol{v}\rightarrow A$) introduces new bound variable $\boldsymbol{v}$ universally quantified. Fig.9 also shows new semantic rules. The first rule is changed so as to yield empty set rather than element **None**. Other three old rules have left unchanged. Two new rules describe semantics of the two new constructors. Note that the value of @-tree may be infinite set. In practice, however, only finite sets are produced.

S-tree ::= …
| (& *S-tree*$_1$ … *S-tree*$_\mathrm{n}$) (finite union, n≥0)
| (@ $\boldsymbol{v}$ → *S-tree*) (infinite union)

$\mathcal{F}$[**None**,$\boldsymbol{V}$]J = {}
…
$\mathcal{F}$[(& A_1 … A_n),$\boldsymbol{V}$]J = [A_i,$\boldsymbol{V}$]J
$\mathcal{F}$[(@$\boldsymbol{v}$ → A),$\boldsymbol{V}$]J = [A,$\boldsymbol{V}$++$\boldsymbol{v}$]J++j

Fig.9. Syntax and Semantics of Multi-Valued Selection Trees

We represent the inverted graph as a collection of multi-valued trees T_w, one for each write operation w on array `X`, which represents a function that maps parameter vector P and iteration vector I_w to the set of respective read operations $\{(r,I_r)\}$.

To construct the source-to-destination graph we need to invert the destination-to-source graph built so far. Another two operations on trees are used here.

Operation $inverse((r,\boldsymbol{V_r}),C_r,T_r)$ takes as input the form (4) and returns a collection of elements of the form

$$((w,\boldsymbol{V}_w)\ T_w), \tag{5}$$

one for each non-**None** term $w\{e_1,\ldots,e_k\}$ of tree T_r. Actually, for each such term a set of linear restrictions on that branch together with equalities $e_i=V_{w,i}$ is taken, of which

equality part is then resolved with respect to variables V_r as a System of Linear Diophantine Equations (SLDE). All accompanying inequalities are then rewritten by eliminating variables V_r and put together in the form of multi-valued tree T_w. All non-**None** terms of that tree use name r. In case the solution space of SLDE is infinite the respective number of @-nodes is created. Also, the SLDE-solver may produce new division nodes. Several elements of the form (5) with the same name w are joined together within new &-node T_w. Finally, the unified element of resulting graph $\mathbf{G}_\mathrm{P}$ which is built in the form

$$((w,\boldsymbol{V}_w)\ C_w\ T_w), \tag{6}$$

where $C_w = (c_1,\ldots,c_K)$ is a list of restrictions that describe iteration space (a set of possible instantiations) of operation w.

Operation *inverse* may be generalized to be applied to the whole graph. Also, it is possible to apply *inverse* to multi-valued trees. This allows for self-verification: apply inverse twice and compare result and original graphs for equivalence.

Operation *simplify-tree*(C,T) is intended to globally optimize the representation of multi-valued tree T under condition list C so as to minimize its complexity as much as possible. In addition, for each @-node, upper and lower bounds of variable v are computed (in the form of affine expressions or a list of such expressions). Thus, the resulting trees T_w are prepared to be effectively compiled into DFL.

There is also a special case of operation *simplify-tree* which is used when argument is known to be a single-valued tree in fact. It strives to eliminate &- and @-nodes.

Note that the inverted graph will also contain trees for names like `A_init` which describe all uses of input arrays. If for non-parameter array such a tree appear non-**None**, it may be regarded as a signal of possible error of type *'attempt to read undefined array element'* with exact description of the error conditions.

9 Transforming Graph to DFL

Suppose we have built the source-to-destination graph $\mathbf{G}_\mathrm{p}$ for some program `P`. The simplest way of building DFL-code is to make a node for each write operation, or assignment statement. Consider an arbitrary assignment w

$$\mathrm{X}x\text{:}\qquad \mathrm{X}(e_1,\ldots,e_k) = E(r_1,\ldots,r_\mathrm{p}) \tag{7}$$

to the element of k-dimentional array X ($k\geq 0$) of type t, which is enclosed in a loop nest with loop variables $\boldsymbol{v_1}$, …, $\boldsymbol{v_l}$ ($l\geq 0$). We assume that each assignment is supplied with a unique label (Xx). Let the right hand side E contain $p\geq 0$ occurrences of different read operations $r_1,\ldots,r_\mathrm{p}$ of types $t_1,\ldots,t_p$, each having the form

$$\mathrm{Y}(\mathrm{f}_1,\ldots,\mathrm{f}_m), \qquad (m\geq 0)$$

For the statement w the graph contains element of the form (6). The general view of the DFL-node is shown on Fig.10. The label of assignment becomes the name of node. Each read operation in the right hand side becomes a separate port with some standard name. Enclosing loop variables form the context of the node.

The first statement of the routine evaluates expression E in which port names are used instead of read operations. $\mathcal{P}(T_w)$ is the translation of multi-valued selection tree T_w to Pascal, which responds for sending out the computed value `a`. Rules of

translation are shown on Fig.11. Each non-**None** term $r\{h_1,\ldots,h_s\}$ of T_w produces send statement `a->`W_r`.aj{`h_1`,…,`h_s`}` where W_r is the label of assignment containing the read *r*, and *j* is the number of respective array access in the right hand side of the assignment (thus `aj` is the respective port name).

```
node Xx(a_1:t1,…,a_p:tp) {v1,…,vl}
var a:t;
begin
    a:=E(a1,…,ap);
    P(Tw)
end
```

Fig.10. The general view of translation for assignment (7)

The tree as a whole becomes a control structure providing necessary activation conditions for these send statements. Note that the translation of division node is improper here as it `uses` **`div`** and **`mod`** that produce incorrect result for negative values of *e*. Translation for @ uses upper and lower bounds (*l*,*u*) of variable ***v*** computed and inserted by operation *simplify-tree*. However the general case is more complex: this tree vertex generally has the form $(@\boldsymbol{v}(l_1\ u_1)\ldots(l_k\ u_k)\rightarrow T)$ and the total range for variable ***v*** is defined as the union of ranges $(l_i\ u_i)$. Hence *min* and *max* should be used to compute true *l* and *u*.

$\mathcal{P}$ (**None**)	(empty statement)
$\mathcal{P}(r\{h_1,\ldots,h_s\})$	`a->`W_r`.aj{`h_1`,…,`h_s`}`
$\mathcal{P}(c \rightarrow T_1{:}T_2)$	**if** $\mathcal{P}(c)$ **then** $\mathcal{P}(T_1)$ **else** $\mathcal{P}(T_2)$
$\mathcal{P}(e{=:}m\boldsymbol{u}{+}\boldsymbol{v} \rightarrow T)$	**begin** $\boldsymbol{u}{:=}e$ **div** m; $\boldsymbol{v} := e$ **mod** m; $\mathcal{P}(T)$ **end**
$\mathcal{P}$ (& T1 … Tq)	**begin** $\mathcal{P}(T_1)$; … ; $\mathcal{P}(T_q)$ **end**
$\mathcal{P}(@\boldsymbol{v}\ (l\ u) \rightarrow T)$	**for** $\boldsymbol{v}{:=}l$ **to** u **do** $\mathcal{P}(T)$

Fig.11. Rules for translation of multi-valued selection tree to Pascal

Our translation is based on the idea that execution of affine program can be replaced by execution of respective set of assignment statements in a feasible order. The order is feasible if it agrees with dataflow dependences: if statement *x* produces value for statement *y* then *x* executes before *y*. Hence, the activation order in data flow model is feasible. On the other hand, each statement will eventually be executed as all its predecessors have been executed and their results sent out.

One problem remains with statements whose nodes have no ports, for example `X[i]=0`. In order to make such *poor* node to execute we add to it a special port of type *any* and assure to send a token to. All poor nodes within a loop body are activated from additional node declared out of the loop. In other words its context is one element shorter. The node's routine contains a loop simulating the original loop. In the loop body tokens are sent, one for each poor inner node. The additional nodes are activated in the same way. On the top level a poor node is activated by sending a token from the outside.

Also additional node is needed for each input array. For *k*-dimensional input array `X` of type `t`, the node `X_in` is generated which has the context of size *k* and one port of type `t`. Each input array element is sent from outside to respective node instance. The node routine sends out the value according to the element of the graph named `X_init`.

For each *k*-dimensional output array `Y` of type *t* the node `Y_out` is generated with single port of type *t* and context of size *k*. Any token sent to it is automatically directed to outside.

All fixed parameters are declared as **`vconst`**. Their values are loaded from outside. The code that loads constants, sends inputs and receives outputs is generated as a separate routine that replaces the body of original subroutine. This code is executed in the main program on the host machine.

10 Example program

Consider the following illustrative example (Fig.12). Here the first loop rewrites input array B into the wider internal array X, in which it is then summed up in pairwise manner. The resulting DFL code is shown on Fig.13.

```
SUBROUTINE SUMMA (B,N,sum)
REAL(8) B(N), X(N*2-1),sum
DO I=1,N
   X(I)=B(I)
ENDDO
DO I=1,N-1
   X(N+I)=X(2*I-1)+X(2*I)
ENDDO
   sum=X(N*2-1)
END
```

Fig.12. The summation program in Fortran

Note that the generated code looks much like that shown on Fig.1. Some local optimizations are needed here. The main procedure is further compiled into code that replaces the original subroutine SUMMA executed on the host machine. Using the same parameter list it sends out elements of input arrays to dataflow process and receives back the result values directly into out parameters.

The translation of this example took 1.14 seconds on PC. We tried a lot of other example programs with no more than 10-20 lines of text (matrix multiplication, LU-factorization, 2d-Jacoby, etc.). For sensible programs the compilation time normally did not exceed 10 seconds. However, for some synthetic tests with very few lines but complex index pattern the time could increase to several minutes and more. The most time-consuming stage is usually the simplification of resulting trees.

```
MODULE SUMMA;
type F1 = {i1:int[4]};
vconst N:int=0;

proc MAIN (B[N]:real, N:int, SUM:real) {}
var q_0:int;
begin
  for q_0 := 1 to N do
    send B[q_0] to B_in{q_0};
  receive SUM;
end;

node B_in(B:real) F1{q_0};
  send B to X_1.B_1{q_0};

node X_1(B_1:real) F1{I};
  if (N=1) then
    send B_1 to SUM_out
  else
    if (((I+1) mod 2)=0) then
      send B_1 to X_2.X_1{((I+1) div 2)}
    else
      send B_1 to X_2.X_2{(I div 2)};

node X_2(X_1:real,X_2:real) F1{I};
  if ((I-N)=(-1)) then
    send (X_1+X_2) to SUM_1.X_1
  else
    if (((I+(N+1)) mod 2)=0) then
      send (X_1+X_2) to X_2.X_1{((I+(N+1)) div 2)}
    else
      send (X_1+X_2) to X_2.X_2{((I+N) div 2)};
```

Fig.13. The DFL code generated for subroutine SUMMA

11 Related work and conclusions

This paper is related to work in three different areas: in dependence analysis as it is done in optimizing and parallelizing compilers, in transforming sequential affine programs to other computational models and in dataflow architectures that were being popular in 80-s.

The foundations of dependence (data flow) analysis for arrays have been well established in 90-s by Feautrier [6,8], Puch[7], Maydan et al. [9], Maslov[10]. Their methods use Omega test and Integer Programming libraries and, in principle, allow to get exact solution for dependence for any pair of read and write reference in affine program. However, for the purpose of parallelization, they were usually applied only to check if dependence exists, not to get the complete description of dependence graph. In our analysis we aim from beginning to the full description of dataflow graph. We use selection trees not only to represent the result of analysis, but also define a set of operations on such trees which hide inside all sophisticated integer algorithms and thus simplify the explanation of program analysis. Our novel method of building graph involves ascendant process of building effect trees and then a descendant process of building state trees. These both use only information on write operations. Then we hit each read operation against respective state tree and get all dependences as a whole source function for this read. Finally, the dependence graph is inverted to obtain all dependences for each write operation.

There is a hot interest to the problem of transforming ordinary programs to other models of computations, especially in conjunction with parallelization. We use the dataflow model of computation as the target. This model has been very popular in the 80's, but later all attempts to promote it failed. To our opinion, the reason of the failure was people's attitude to dataflow as to a hardware model intended for

execution of ordinary control flow programs. The problem of translating arbitrary control flow program to data flow was considered accordingly [11] as a dataflow simulation of sequential execution over the memory. Minimal reordering is allowed that does not violate normal order of reads and writes.

In contrast, we consider dataflow as an independent model of programming and do not even intend the dataflow language to mimic ordinary programming languages (as Id or Sisal do). The key feature of our incarnation of the dataflow idea is the way we use context; it is an object of total programmer's control that is used primarily for addressing. Usually, the programs should be totally redesigned. In fact, there exist a lot of program examples written in our model of computation that radically differ from their habitual view.

There exists also a bulk of papers in which the same class of affine programs or its minor extensions is translated into other computation models. In [12] process networks (PN) are used as a target programming model. To accomplish the translation their compiler needs to extract more information than ours; specifically, they have to statically order the data streams between processes, whereas in our model all necessary correspondence is provided at run time on the base of iteration indices supplied within token' tags. Also, in [2] a code for CUDA is generated with the aid of additional extraction (from the standard polytope model) of multi-level tiled parallel schedule. In contrast, we only build the inverted form of the dataflow graph of the source program (which is in fact very similar to Z-polyhedral model [13]) and just treat that graph as a code for dataflow machine. We needn't generate any parallel schedule at compile time. All actual parallelism is extracted on the fly by runtime system which must be implemented in hardware for good performance [14,15].

11 Future work

So, the time schedule is not needed to be provided statically for our computation model. However, the compiler *must* provide the space schedule or distribution of virtual computation space across physical processor space. The distribution function is a (usually affine) function that maps the total virtual address space of node instances onto (probably multi-dimentional) space of processor elements (PE) numbers. This function must be provided either by the user (in terms of source iteration space) or generated by the compiler (may be with user hints) . The two following criteria must be satisfied:

- the workload is uniformly balanced;
- the amount of communication is minimized.

At present, only a strictly affine programs are allowed. Moreover, the whole program unit (subroutine) must be affine. We plan to lift some of restrictions and to allow for

- non-affine conditional expressions in if-statements;
- non-unit step in do-loops;
- using whole division in affine expressions;
- some local variables in affine expressions;
- using affine subprograms with side-effects.

The work was supported by Russian Academy of Sciences Presidium Program for Fundamental Research No.15 “Fundamental Problems of System Programming” in 2009-2011.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley (2006).
2. Baskaran, M.M., Ramanujam, J., Sadayappan, P.: Automatic C-to-CUDA Code Generation for Affine Programs. In: Gupta, R. (ed.) CC 2010. LNCS, vol. 6011, pp. 244-263, Springer, Heidelberg (2010).
3. Voevodin, V.V., and Voevodin, Vl.V.: Parallel Computations (Параллельные вычисления). BKhV-Peterburg, St. Petersburg (2004) [in Russian].
4. Klimov, Ark.V.: The Use of Selection Trees for Describing States in Parallelizing Compiler. In: Proceedings of All-Russian Scientific Conference “Scientific service in Internet”, pp.238-240 MSU (2009) [in Russian].
5. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In: Comptes Rendus du I congrès de Mathématiciens des Pays Slaves, pp. 92–101, Warszawa (1929),.
6. Feautrier, P.: Parametric Integer Programming. In: RAIRO Recherche Opérationnelle. 22:243-268 (September 1988).
7. Pugh, W.: The Omega Test: a fast and practical integer programming algorithm for dependence analysis. In: Proceedings of the 1991 ACM/IEEE conference on Supercomputing ACM New York, NY, USA (1991).
8. Feautrier, P.: Dataflow Analysis of Array and Scalar References. In: International Journal of Parallel Programming, V 20, N 1, pp. 23—53 (1991).
9. Maydan, D.E., Hennessy, J.L., Lam, M.S.: Efficient and Exact Data Dependence Analysis. In: ACM SIGPLAN’91 Conference on Programming Language Design and Implementation, pp.1-14 (June 1991).
10. Maslov, V.: Lazy Array Data-Flow Dependence Analysis. In: Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 311-325 (January 1994).
11. Beck, M., Johnson, R., Pingaly, K.: From Control flow to Dataflow. In: Journal of Parallel and Distributed computing, V 12 Issue 2, pp. 118-129 (June 1991).
12. Turjan, A., Kienhuis, B., Deprettere, E.: Translating affine nested-loop programs to process networks. In: Proceedings of the International Conference on Compiler, Architecture, and Synthesis for Embedded Systems, pp. 220-229, Washington D.C., USA, (Sept. 2004).
13. Gautam Gupta, Rajopadhue, S.: The Z-polyhedral model. In: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, ACM New York, NY, USA (2007).
14. Burtsev, V.S.: “Vybor novoj sistemy organizacii vypolneniya vysokoparallel’nyh vychislitel’nyh processov, primery vozmozhnyh arhitekturnyh reshenij postroeniya superEVM” (The choice of a new organization system of execution of highly-parallel computation processes and examples of possible supercomputer architecture solutions), // In: Burtsev, V.S. Parallelizm vychislitelnyh processov i razvitie arhitektury superEVM, IVVS RAS, Moscow, pp. 41-78 (1997) [in Russian].
15. Stempkovsky, A.L., Levchenko, N.N., Okunev, S.A., Tsvetkov, V.V.: Parallel dataflow computing system – the further development of architecture and the structural organization of the computing system with automatic distribution of resources. In: “Informatsionnye tekhnologii”, N 10, pp. 2–7 (2008) [in Russian].