— LooPo —
Loop Parallelization in the Polytope Model

Nils Ellmenreich, Peter Faber, Martin Griebl,
Robert Günz, Harald Keimer, Wolfgang Meisl,
Sabine Wetzel, Christian Wieninger, Alexander Wüst

April 15, 1996

1 Introduction

LooPo is a project at the Chair for Programming at the Department of Mathematics and Computer Science of the University of Passau. It is a prototype system to assist us in the research on and evaluation of space-time mapping methods for loop parallelization. To that end, it implements the complete path from executable source code to executable target code, with switches for choosing alternative methods. LooPo is part of the DFG-funded project *RecuR* (*Regular Concurrency in Recursions*).

The first phase, an implementation of space-time mapping methods for nested for loops [Len93], started in late 1994. The second phase is due to start in mid 1996 and will give LooPo the capability of dealing with while loops in the loop nest [LG95].

With Martin Griebl as the project manager, up to now, eleven students have been involved in the implementation of different modules. The LooPo team can be contacted by email at `loopo@fmi.uni-passau.de`. There is a web-page for more information about LooPo at `http://www.uni-passau.de/~loopo`.

The current version (1.0) of LooPo including the source code in C++ and the front end in Tcl/Tk is available under the general public gnu license. LooPo will only use freely available software to enable a wide-spread use. LooPo runs on Sun workstations under SunOS 4.1.x and Solaris 2.x, and on PCs under Linux. For instructions on how to install LooPo see Section 4.

2 Theoretical Background

The space-time mapping methods we study are based on the polytope model of nested loops, and on some extensions of it.

The polytope model is a useful model of computation for the static parallelization of nested for loops [Len93]. The model represents the atomic iteration steps of d perfectly nested for loops as the points of a polytope in $\mathbb{Z}^d$; each loop defines the extent of the polytope in one dimension. The faces of the polytope correspond to the bounds of the loops; they are all known at compile time. This enables the compile-time discovery of maximal parallelism—relative to the choices available within the method, which are limited by the data dependences of the source program and by the requirement that the space-time mapping defining the parallelism be affine.

2.1 The basic polytope method

Parallelization in the polytope model proceeds as follows (Figure 1).

First, one transforms d perfectly nested source loops into a d-dimensional polytope, where each loop defines the extent of the polytope in one dimension. We call this polytope the *index space*. Each point of the index space represents one iteration step of the loop nest. The coordinates of the point are given by the values of the loop indices at that step; the vector of these coordinates is called the *index vector*.

Then one applies an affine coordinate transformation, the *space-time mapping*, to the polytope and obtains another polytope in which some dimensions enumerate space and the others enumerate time. We call the transformed polytope the *target space*. The standard techniques for space-time mapping in the polytope model require uniform dependences [KMW67]. In this

```
for i := 0 to n do
    for j := 0 to i + 2 do
        A(i, j) := A(i − 1, j)
                   +A(i, j − 1)
    end
end
```

```
for t := 0 to 2n + 2 do
    forall p := max(0, t−n) to min(t, ⌊t/2⌋ + 1) do
        A(t−p, p) := A(t−p−1, p)
                     +A(t−p, p−1)
    end
end
```

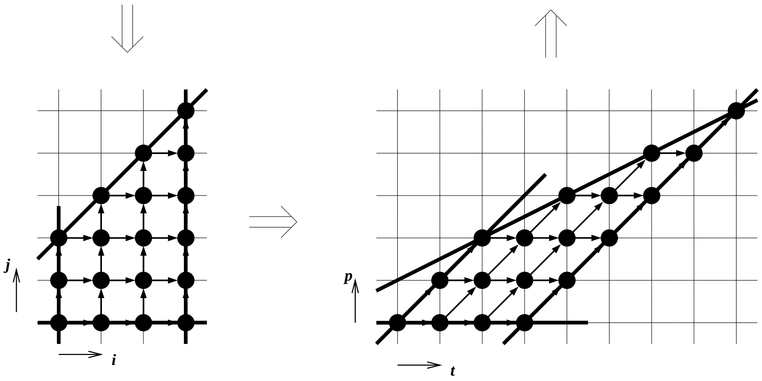

Figure 1: Parallelization in the model

case, all points of the index space have identical dependence vectors which are independent of the problem size. LooPo accepts any affine, not only uniform dependences.

Finally, one translates this polytope back to a nest of target loops, where each space dimension becomes a parallel loop and each time dimension becomes a sequential loop.

2.2 Extension to affine-by-statement scheduling

The basic model represents only perfectly nested loops. This severe restriction can be relaxed by applying the basic method to every single statement separately rather than to the body as a whole. Thus, every statement has its own index space, index vector, space-time mapping and target space [DR94, Fea92a, KPR94, Rao85]. An *operation* in the program is identified by a statement together with its index vector. Of course, the feature of statement-wise space-time mapping complicates the generation of target code significantly (cf. Section 3.8).

3 The Structure of LooPo

Like most compilers, LooPo consists of a number of modules which transform the source program to an executable target program. There are modules for scanning and parsing, dependence analysis, scheduling, allocation, target loop nest generation and target code output. There is also a graphical displayer (*Dispo*) of index spaces and iteration dependence graphs of loop nests.

Each of these modules reflects a self-contained part of the project and resides in a separate binary file. A front end provides a graphical interface by which he/she can control all modules of LooPo. In order to optimize LooPo's results and suit the needs of the user, most modules provide an option window where specific features can be altered.

Communication between modules is usually realized by data exchange via files in a temporary directory whose location is being displayed in the top-level window. Interaction between the front end and the other modules takes place by uni-directional FIFO communication in one direction and by command line options in the other.

In the sequel detailed information about the modules and their usage is being presented.

3.1 The Front End

3.1.1 Purpose

The front end consists of a binary file, compiled from C++ sources, and a Tcl-script which is loaded at run time. It gives the user control over how and when to run LooPo modules.

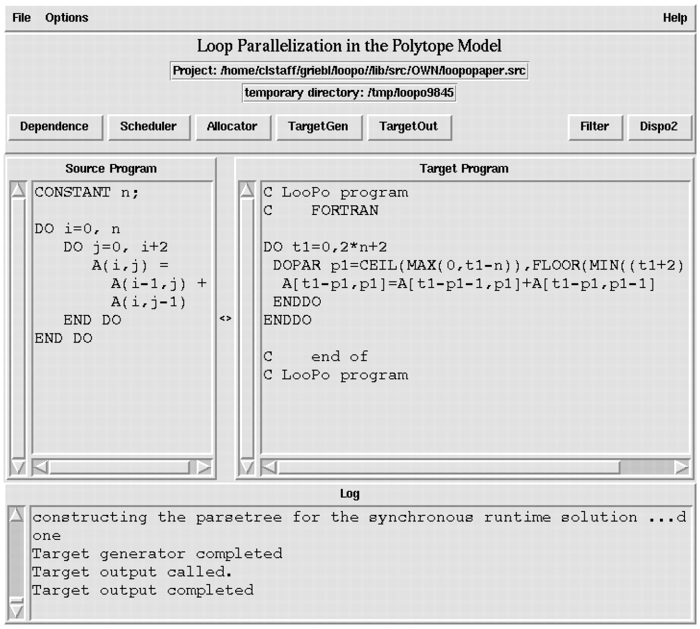

Figure 2: The front end of LooPo

A LooPo session starts with selecting a source file which is automatically passed on to the scanner/parser module. The selection can be made by choosing the entry "Set project name..." in the File menu. The status of currently set options may be saved via another entry of the File menu and is from then on being loaded at startup time. The options are saved in `~/.looporc`. The next menu is called Options containing several windows to select module-specific options, which are described in the respective module sections.

Below the menu bar the user finds buttons to invoke modules. The required order is from left to right, with the following exceptions:

- The dispo and filter module may be called after a successful run of the dependence module.
- The allocator may be called before the scheduler.
- The allocator may be skipped in the case of the Lamport's scheduling method (which integrates an allocation method; cf. Section 3.4).

Feedback about the modules computations, possible errors and finally the report of their completion are displayed in the log at the bottom of LooPo's main window. More detailed information appears in special output windows which are displayed after the modules completion, whether successful or not. The general options window offers a choice of whether the output windows should start iconified or not.

Of the middle area of the main window, the left part contains the source program, the right part the parallelized target program after a successful run of the targout module.

In addition to the front end binary, some object files are collected in the so-called *dialogs* library, which is linked to most other modules and provides C++ classes for help- and error-dialog boxes and defines the interface for communication via the FIFO.

A detailed description of the front end can be found in [Ell95] (text in German).

3.1.2 Input

Only the saved options in `~/.looporc` may be considered as input for the frontend module.

3.1.3 Options

None.

3.1.4 Output

The file `~/.looporc` may be saved from within the File menu.

3.2 The Scanner/Parser

3.2.1 Purpose

The Scanner/Parser module is automatically invoked after selecting a source program within the front end module. Its main task is to create a parse tree and a symbol table from the given program. In addition, user-defined dependences (if any) are stored in internal data structures. Several programming languages can be used in the program, like C or FORTRAN. One can even mix different programming languages in the source program statement by statement.

3.2.2 Input

The Scanner/Parser module reads the file <*input-file*> selected in the front-end to create the symbol table, parse tree and dependence list of user-defined dependences. The grammar for the input language is presented in Appendix A. An additional restriction is that the current version of LooPo only deals with nests of `for` loops (possibly imperfect nests), whose bounds are affine expressions in outer loop indices and structure parameters.

3.2.3 Options

None.

3.2.4 Output

The Scanner/Parser module writes the file <*input-file*>`.out` which will be referred to as <*project*> in the rest of this documentation. This file contains the symbol table, the parse tree and the dependence list for user defined dependences (if any) in this order. For many of the other LooPo modules this file is used for input.

All these files are placed in the directory `/tmp/loopo`$<pid>$, where $<pid>$ is the process id of LooPo. We shall call this folder <*temp_dir*>. Sometimes we need to refer to the installation directory of LooPo as <*LooPoDir*>.

Further information about the Scanner/Parser module can be found in [Gün96] (text in German).

3.3 The Dependence Test

3.3.1 Purpose

This module calculates the data dependences occurring in the source program. The analysis is based on the method by U. Banerjee [Ban93], but extends it in several aspects. The most important extensions are:

- The dimension of the non-empty polytope describing all integer solutions for the vector t is no longer restricted to 0 or 1. It is now possible to handle polytopes of any dimensionality.
- The loops need not be nested perfectly; imperfectly nested loops can be handled.
- All methods can deal with structure parameters.

The tool computes the set of all memory conflicts representing them by the so called *h-transformation* which was introduced by P. Feautrier [Fea91]. Banerjee's methods do not calculate the precise flow of data but compute all pairs of operations with conflicting accesses to the same memory cell. Therefore, it might become necessary to introduce new parameters (*pseudo parameters*) in order to represent sets of h-transformations. A detailed description of the analysis can be found in [Kei96] (text in German).

3.3.2 Input

The dependence module reads the <*project*> file generated by the scanner/parser.

3.3.3 Options

External tool: decides whether PIP [Fea88] or Fourier-Motzkin [Ban93] shall be used to solve parametric integer problems.

Speedup: avoid calling the external tool whenever possible.

Delete obsolete dependences: if set, all obsolete dependences (e.g., dependences with empty source/target spaces) will be deleted.

Renumber dependences: if set, the list of dependences will be renumbered after the calculation is finished (some dependences might be deleted by making use of the delete option as described above).

Display additional warnings: if set, the dependence module will perform an enhanced set of tests and report several warnings (e.g., if there are some distance vectors among the user-defined dependences which might become negative).

Verbose: if set, the user will get detailed information about what is going on during the calculations; otherwise only the result will be displayed.

Do not touch user-defined dependences: if set, the user-defined dependences will *not* be touched, even if this might be necessary in order to guarantee legal distance vectors.

Handle only user-defined dependences: if set, the module takes only the user defined dependences into account; no dependences will be calculated by the tool itself.

3.3.4 Output

The dependence module produces nine output files; the extensions of these files are defined in <*LooPoDir*>`/general/globaldefs.h`

<*project*>`.ispc`: describes the index spaces of all statements in the main part of the source program.

<*project*>`.dep`: contains the list of all dependences as the main output of the dependence module.

<*project*>`.fil`: contains the list of all dependences; this file is identical to the `.dep` file and must be written because the dispo module might be invoked without running the filter module before (cf. 3.10).

<*project*>`.grp`: contains all statement groups; statements in the main part of the source program, which are surrounded by the same loops and also have the same loop level, belong to one statement group; the groups are needed by the filter module.

<*project*>`.dpmsg`: the message file which holds the entire output of the dependence module; its name is given to the front end module using the FIFO communication (cf. Section 3) which will display the content when the dependence module has finished.

<*project*>`.acp`: contains the ids of nodes in the parse tree representing the accesses that caused a dependence.

<*project*>`.acc`: contains the entire list of all accesses in the source program.

<*project*>`.dpbak`: contains a backup of the scanner/parser output.

<*project*>: contains the modified parse tree after inserting pseudo-parameters.

3.4 The Lamport Scheduler

3.4.1 Purpose

The hyperplane method of Leslie Lamport [Lam74] only deals with perfectly nested programs and uniform dependences. It yields a one-dimensional affine schedule for the complete loop body and as allocation a projection onto the source axes such that the complete transformation (schedule and allocation together) is unimodular.

3.4.2 Input

Index spaces: One index space for each statement is required.
They are read from <*project*>`.ispc`.

Dependences: A list of all dependences in the program.
They are read from <*project*>`.dep`.

Both input files are written by the dependence module.

3.4.3 Options

The original hyperplane method is called if the option **linear** is selected, whereas **affine** translates the schedules so as to assure that schedules start at time 0.

3.4.4 Output

- The schedule is saved in <*project*>`.sched`.
- The allocation is saved in <*project*>`.place`.
- The text for the output window is saved in `schedalloc_output`

3.5 The Feautrier Scheduler

3.5.1 Purpose

The Feautrier scheduler yields the optimal (concave) schedule for imperfectly nested loops with affine dependences, at the cost of a longer computation time based on the necessity of dealing with parametric integer linear programming. The resulting schedule for every statement can be multidimensional and piecewise affine [Fea92a, Fea92b].

3.5.2 Input

See Section 3.4.2.

3.5.3 Options

The original method **by statement** schedules statements separately. For comparability with Lamport's method one can call the Feautrier scheduler **by iteration** (in the case of a perfectly nested input program), which enforces the same schedule for all statements in the loop body.

3.5.4 Output

The schedules are saved in <*project*>`.sched`. The text for the output window is saved in `schedalloc_output`.

3.6 The Darte-Vivien Scheduler

3.6.1 Purpose

The Darte-Vivien module is a fast scheduler with reasonably good results. [DV94] is able to schedule arbitrary loop programs with uniform and non-uniform dependences. It uses a less precise dependence description than the Feautrier scheduler does. Therefore the results of this module are settled somewhere between those of the Lamport and the Feautrier schedulers. For a more detailed description refer to [Mei96].

3.6.2 Input

See Section 3.4.2.

3.6.3 Options

Dependence Conversion:

Dependences are converted from affine functions as read from the input file to sets of distance vectors. Since this conversion is not precise, the modeled set of distance vectors is a superset of read dependence vectors.

show results: Show the result of the conversion in the output window. This is always just one line per dependence for the standard method and may be a couple of lines for the cone method. The following notation for components of the dependence vectors is used:

x	only value x
$x+$	all values greater or equal than x
$x-$	all values lower or equal than x
$*$	all values

Scheduling Method:

avoid distribution of statements: Try to avoid constants in the first dimension of a statement's schedule. In many cases this leads to more compact schedules (default).

only non-negative coefficients: The coefficients in the schedule will be strictly non-negative. This is a real restriction, so do not use it if you do not need it (intended for internal use).

shift schedules: With this option enabled, the schedules will be non-negative in all dimensions (default).

3.6.4 Output

This scheduler module writes one output file <*project*>`.sched`, which contains the piecewise affine schedules for each statement. The text for the output window is saved in `schedalloc_output`.

3.7 The Feautrier Allocator

3.7.1 Purpose

Feautrier's allocation method [Fea94] determines the placement of operations on virtual processors. It is based on the owner computes rule and tries to "cut" dependences by mapping the depending operations on the same processor, starting from dependences in the highest dimensions (greedy heuristics). The allocator does not use the schedule, thus possibly generating linearly dependent allocation dimensions.

3.7.2 Input

See Section 3.4.2. Additionally, the file <*project*>`.sched` is read to determine the dimensionality of the schedule.

3.7.3 Options

Allocator dimensions: One either chooses a fixed dimensionality greater than 1 for the allocation or, in the case of 0, the dimensionality is set to the maximal dimensionality of the index spaces reduced by the dimensionality of the schedule (default dimensionality).

Statements with insufficient dimensions: can either be skipped in the allocation process, i.e., their allocation does not get a higher dimensionality than their default dimensionality, or their allocations are computed until the desired dimensionality is reached (even by filling up with zero entries, if necessary). The latter guarantees the same dimensionality for all statements.

Method by statement/by iteration: as in the Feautrier scheduler.

Sort dependences by communications value: enables the greedy default method. The allocator uses the order of dependences in the input file if this option is switched off.

3.7.4 Output

The allocations are saved in <*project*>`.place`. The text for the output window is saved to `schedalloc_output`.

3.8 The (Raw) Target Generator

3.8.1 Purpose

The target generation module constructs the parse tree of the transformed source program. It takes the index spaces, schedule and allocation to build transformation matrices for every statement in the source program. Since the generated parse tree does not contain any synchronization or communication statements this module should be called "raw target generator".

This process takes place in three steps. First, all required information is loaded. Second, the source statements are transformed individually. In order to achieve that, the transformation matrices are computed, the target index spaces are constructed, strides and offsets are calculated, and finally all expressions in every statement are re-indexed according to its transformed coordinates. Third, the transformed statements in their respective index spaces (*parts*) are merged to a single target program. Aside from the two options of synchronous or asynchronous code, three merging strategies are available (see [Wet95] for details):

1. The parts are simply combined with a *parallel operator*, i.e., there are several separate loop nests which are assumed to be executed in parallel.

2. All index spaces of the parts are merged to one index space to construct a perfect loop nest. The statements inside the body are guarded by `if`s to determine at run time whether or not they are to be executed. This is the *run time* solution.

3. To avoid `if`s at run time, one determines at compile time precisely in which regions which statements must be executed, and enumerates these regions successively. This is only implemented for the outermost dimension, the others are merged by the run time method. Problem: for n parameters in the loop bounds, an $O(n!)$ case distinction in the target program is necessary to determine their relative order.

3.8.2 Input

schedule and allocation: as computed by the scheduler/allocator modules and stored in <*project*>`.sched` and <*project*>`.place`, respectively. They may be piecewise affine and differ from statement to statement. Schedule and allocation may contain linearly dependent rows and may map to an index space of higher or lower dimension than the initial one.

index spaces: one index space for each statement is required. They are read from <*project*>`.ispc`.

source parse tree: the parse tree in the file named <*project*> contains the original statements, which will be transformed by this module in order to generate the new parse tree.

3.8.3 Options

generate synchronous loops: make the time loop(s) outermost to generate a synchronous program.

generate asynchronous loops: lay out the outermost loop(s) in space to generate an asynchronous program.

solution with parallel operator: (see description in section 3.8.1, strategy 1.)

solution at run time: (see description in section 3.8.1, strategy 2.)

solution at compile time: (see description in section 3.8.1, strategy 3.)

Note that not all combinations of the options above are possible.

3.8.4 Output

new parse tree: describing the transformed program (*<project>*`.parse`) for shared memory machines and without synchronisation statements.

transformation matrices: computed intermediately, defining bijective transformations from the original index spaces to the index spaces of the target program (*<project>*`.trafo`).

3.8.5 Known Bug

Any fraction p/q in the output of a scheduler is meant to be $\lfloor p/q \rfloor$ but the target generator treats them as real fractions, using strides smaller than one.

3.9 The Target Output

3.9.1 Purpose

The target output module targout is used to print the parse tree generated by targgen in a language chosen from a variety of possible output languages. These are described in so-called *language description files* found in the LooPo library directory. The main task of the targout module is to produce output that may (after re-establishing a complete program from the program part processed by LooPo) be compiled into a binary for a PARIX[1]-machine.

As the targgen module does not consider the need for sending and receiving data in a distributed memory model or the need for synchronisation of any kind, these statements are produced by targout, if indicated by the `comm`- or `sync`-option, respectively (see below).

A detailed description of an earlier version of the targout module can be found in [Fab95] (text in German).

3.9.2 Input

<project>`.acp`**:** the access pairs, each causing a certain dependence as determined by the dependence module; the access pairs are used to determine which variable has to be sent/received in a distributed memory model.

<project>`.dep`**:** the list of dependences; for each dependence listed here and each copy of statements concerning this dependence[2], a `send/receive` pair is produced, using the *h-transformations* given in this file (modified by transformation matrices which are produced by targgen), in order to determine the sending and receiving processor, respectively.

<project>`.ispc`**:** the index spaces of statements (divided into source and target spaces with respect to a certain dependence); they are used to decide whether a `send/receive` statement really has to be executed.

*<project>***:** the source parse tree is needed to standardize the transformation matrices produced by targgen so that they fit together with the *h-transformations* produced by the dependence module.

<project>`.parse`**:** the target parse tree that is to be printed.

<project>`.trafo`**:** the target transformation matrices produced by targgen which define mappings from the old (source) indices to the new (target) indices introduced by targgen; these mappings are used to produce *h-transformations* for the target program from the *h-transformations* computed by the dependence module.

<LIBDIR>`/`*<language_file>***:** the *language description file* containing information about the syntax of the output language
(The LooPo library directory *<LIBDIR>* is usually *<LooPoDir>*`/lib`, and the canonical form of *<language_file>* is *<language>*`.lang`, where *<language>* is the desired output language.)

[1] PARIX^{TM} is an operating system for massively parallel computers with distributed memory by the PARSYTEC company based on the SPMD approach (see [Par93])

[2] targgen may have produced copies of certain statements

<LIBDIR>/*<renaming_file>*: the *renaming configuration file* containing the preferences about the names given to the target indices introduced by the targgen module (whether or not they are called `loopo_targgen_`*<n>*))
The default value for *<renaming_file>* is `rename.conf`.

3.9.3 Options

Rename loop indices: if set, the indices of the target loops are renamed according to the *renaming_file*; otherwise the automatically generated internal names are used.

Distinguish between `doall` and `doacross`: if set, there will be different `do` statements for parallel loops carrying a dependence (`doacross` loops) than for loops that may be executed in parallel independently from each other (`doall` loops).

Count Statements: if set, the statement number is printed before each statement as a remark.

Test whether enough processors available: if set, there will be an `if` statement before the outermost loop testing, if there are enough processors available to cover the whole index space put up by the parallel loops.

Output of synchronisation statements: if set to *norm*, a `barrier` statement is placed after each parallel loop; if set to *on*, a `barrier` statement is placed before the end of each sequential loop.

Output of communication statements: if set to *on*, `send` and `receive` statements are printed for every dependence; if set to *norm*, `send/receive` statements are only printed if the sending processor may be another one than the receiving processor; if set to *off*, no communication statements are printed.

Output language: the desired output language; currently, the following languages are available:

- `C` – normal `C` code with pseudo functions for parallel loops, `send`, `receive` and `barrier` statements.
- `Fortran` – normal `Fortran` code with pseudo functions for parallel loops, `send`, `receive` and `barrier` statements.
- `Parix` – `C` code that can be compiled including file *<LooPoDir>*`/lib/loopo_comm.h` and linked with (the object file made from) *<LooPoDir>*`/lib/loopo_comm.c` in order to run on a Parix-machine.
- Systolician 1 – a pseudo language described in [Sch95]; its grammar is shown in appendix A (the constructions peculiar to this language bear the prefix `s1`).
- Systolician 2 – a pseudo language described in [Sch95], which may only be used for perfectly nested loops; its grammar is shown in appendix A (the constructions peculiar to this language bear the prefix `s2`).

Channel declaration: if the output language supports communication via channels, these channels may be declared at the beginning of the program part or before the outermost loop in which they are being used; if additional variables have to be declared, they are declared at the same position as the corresponding channels.

Note that there are combinations of options which result in programs that are not executable on a real machine: On a distributed memory system, communication must be set to *norm* or *on*; on a shared memory system, synchronisation must be set to *on*.

3.9.4 Output

<temp_dir>`/target_code`: the target program.

3.10 The Dependence Filter

3.10.1 Purpose

The dispo module (described in Section 3.11) is able to display the source index space and the dependences therein in up to three dimensions. In the current implementation all statements must be in the same index space; this excludes imperfectly nested loops. The filter module provides a graphical interface to enable the user to select a subset of statements which satisfy these restrictions. The set of dependences which stem from the selected statements can again be reduced by mouse selection. The user may save the selected set of dependences for display with Dispo or cancel the process. In the future this module might be incorporated into Dispo.

3.10.2 Input

The filter module reads three files: *<project>*`.ispc` to extract the list of statements, *<project>*`.dep` for the list of dependences and *<project>*`.grp` for information regarding which statements belong to the same group (cf. Section 3.3.4).

3.10.3 Options

None.

3.10.4 Output

The filter module writes the file *<project>*`.fil` which has the same format as *<project>*`.dep` but contains only the subset of filtered dependences. This file is read by the dispo module. In the case that filter is not invoked, the dependence module writes two identical files, *<project>*`.dep` and *<project>*`.fil`. This way, filter can also be invoked several times for the use with Dispo without losing information.

3.11 Dispo

3.11.1 Purpose

The module DISPO2 (*DIsplay System for POlyhedra 2*) supports the users intuition by visualizing the index space of a perfect loop nest up to dimension 3. In addition, dependences between index points are displayed with arrows whose colour denotes the dependence type. If structural parameters are present in the current source program the user has to provide numeric values for them in order to get a polytope with numeric bounds.

Three sets of simple operations may be applied to the displayed polytope: rotations, scaling and translations. This is done by pressing the respective button in DISPO2's main window (Figure 3). The effects of single operations (e.g., the x rotation angle) may be tuned by sliders in the upper right part of the window. Below that one can find the axes of the corresponding coordinate system which is updated with each operation to maintain the user's orientation even after several operations on the polytope have been performed.

Below the operation buttons one has buttons for resetting the display, viewing the target space after having entered a transformation matrix and the source space button to switch back to the source polytope afterwards. To the lower left the print button will save the current image of the polytope as a postscript file. The OK button finally exits the DISPO2 module.

See [Wüs95] for further explanation (text in German).

3.11.2 Input

The DISPO2 module reads three files: *<project>*`.ispc` to extract the index space, *<project>*`.fil` for the list of filtered dependences and *<project>*`.grp` for information regarding which statements belong to the same group (cf. Section 3.3.4).

3.11.3 Options

None.

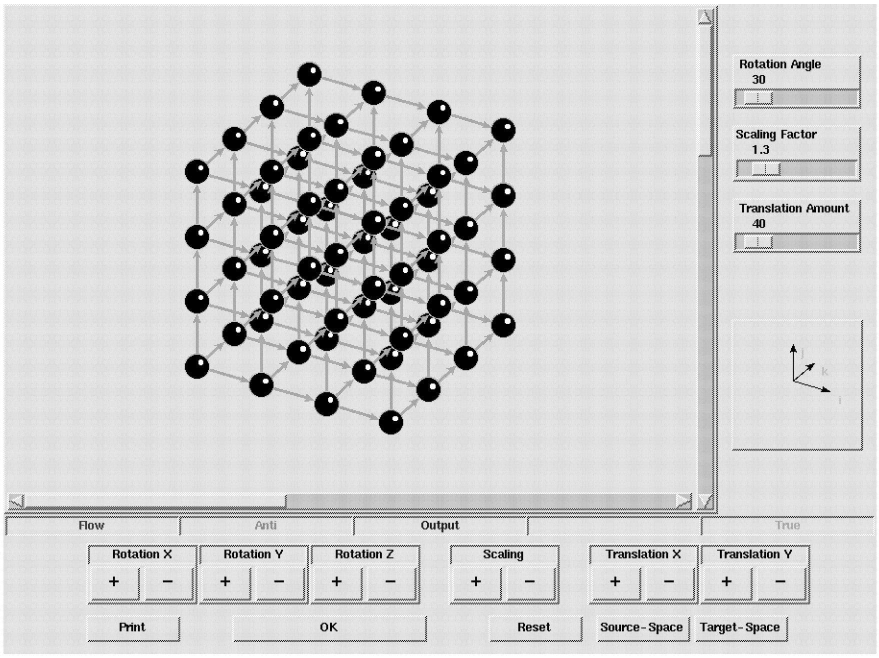

Figure 3: Main Window of Dipso2

3.11.4 Output

None.

4 Installation of LooPo

The development of the LooPo project has been done on Solaris 2.x and several flavours of Linux. In the beginning we started on SunOS 4.1.3; therefore installation support for these three operating systems is provided. But except some basic system calls which are defined by the POSIX standard (e.g, *named pipes*) no special features of these OS's have been used. Therefore it should be no problem to port LooPo to other standard UNIXes; in fact, we'd like to hear from if you did it.

In order to compile LooPo a set of freely available software is required. The numbers in parentheses are the version numbers we currently use which means you may run into trouble if you use older releases (with Tcl/Tk this is certainly the case, whereas with gcc/g++ version 2.6.3 or 2.7.0 it is not).

The required non-standard packages are:

- GNU make (3.74)
- GNU gcc/g++ compiler (2.7.2) together with GNU libg++
- Tcl/Tk toolkit (7.4/4.0)

Needless to say we make use of X-Windows (X11R6). Additionally, the grammars for LooPo's input language are processed by bison++ and flex++ but in the standard distribution we provide the corresponding C++ files so if you don't want to change the grammar you don't need these two tools (but if you want to do, change the variable definitions `BISONPLUSPLUS` and `NOFLEXPLUSPLUS` from in the Makefile.)

All mentioned packages are available from most software archives on the Internet.

4.1 How to install

LooPo comes as a gzipped tar archive. It will create a directory called `loopo-1.0` where the entire source tree resides. Before compiling LooPo you will have to change the Makefile to suit your local environment.

First, you may want to change `LIBDIR` and `BINDIR`. After installation the former will contain Tcl-scripts, examples, documentation and some auxiliary files needed at run time whereas the latter is the location where the binaries will go.

If you do have GNU bash the supported operating systems are automatically determined. Otherwise set the `OS` variable accordingly. In addition you may want to alter names and paths for the required libraries and the GNU make binary.

Next step is to invoke GNU make by typing

```
host$ gmake
```

or whatever it is called on your system. Please note that we do require GNU make; other makes will most likely fail. After a successful compilation you can type

```
host$ gmake install
```

which will copy LooPo into `LIBDIR` and `BINDIR`. If `BINDIR` is in your path you start LooPo by invoking it with

```
host$ loopo
```

If you have problems compiling LooPo on one of the supported platforms, succeeded on other systems or just have some comments please send mail to `loopo@fmi.uni-passau.de`.

5 Integration of new Schedulers and Allocators

The modularity of LooPo makes it easy to plug in a new scheduler or allocator, as long as they produce (possibly multidimensional) piecewise affine functions. In the current version, the owner computes rule is hard coded. This will be changed in the next release, where the variable distribution will also become a multidimensional, piecewise affine function. The file format for such functions is documented in `http://www.uni-passau.de/~loopo/doc/trafo-io-speci.html`.

6 Future Prospects

This release does not contain all modules which are currently being developed. Some work has to be done before the remaining modules can be released at a later date. The list of new modules may include one for treating while loops, another one for eliminating some more dependences by transforming the source program into single-assignment form, and a module for folding and partitioning to map the target code to a machine with fewer processors than required by the allocation.

7 Acknowledgments

Thanks to all the past members of the LooPo team who still take the time to help us, and the present members of the team for the wonderful cooperation. In addition, we all want to thank Jean-François Collard for numerous and very helpful explanations and fruitful discussions. Financial support was received from the Deutsche Forschungsgemeinschaft (DFG) by the project RecuR and from the DAAD by the German-French exchange program Procope.

A Grammar for the Input Language

The grammar can be found in <*LooPoDir*>`/scanpars2/grammar.y`. It is here included for easy reference:

```
program       : definitions statements end_program
              | statements end_program
              | statements definitions
              | definitions end_program
              | definitions statements definitions
              | end_program
```

```
                ;
end_program   : dep definitions
              | dep

              |
              ;
definitions   : definitions definition
              | definition
              ;
definition    : c_prototype
              | f_prototype
              | c_function
              | f_function
              | CONSTANT bloblist ';'
              ;
c_prototype   : c_type IDPAR '(' c_parameters ')' ';'
              | IDPAR '(' c_parameters ')' ';'
              ;
c_parameters  : c_para_list
              | NOTYPE
              ;
c_para_list   : c_para_list ',' c_para
              | c_para
              ;
c_para        : intent IDID argument
              | intent argument
              ;
argument      : ID
              | IDCOM
              | IDPAR '[' ']'
              | IDPAR '[' numlist ']'
              ;
c_function    : c_type IDPAR '(' c_parameters ')'
                c_fct_body
              | IDPAR '(' c_parameters ')'
                c_fct_body
              ;
c_type        : IDID
              | NOTYPE
              ;
c_fct_body    : '{' lokals c_fct_stats '}'
              ;
c_fct_stats   : c_fct_stats statement
              | c_fct_stats RETURN expr ';'
              | c_fct_stats RETURN ';'
              |
              ;
lokals        : lokals list_of_lokals ';'
              |
              ;
list_of_lokals : IDID arg_list
               ;
arg_list      : arg_list ',' arg
              | arg
              ;
arg           : ID
              | IDCOM
              | IDPAR '[' numlist ']'
              ;
numlist       : numlist ',' NUM
              | NUM
              ;

f_prototype   : SUBROUTINE fctname '(' f_parameters ')' fortran
                   para_def_io
```

```
                END SUBROUTINE fctname
              | FUNCTION   fctname '(' f_parameters ')' fortran
                   para_def
                END FUNCTION fctname
              ;
fortran       :
              ;
f_parameters  : f_para_list
              |
              ;
f_para_list   : f_para_list ',' ID
              | ID
              | f_para_list ',' IDCOM
              | IDCOM
              ;
para_def      : para_def IDID f_arg_list
              |
              ;
para_def_io   : para_def_io def_io
              |
              ;
def_io        : IDCOM ',' INTENT '(' intent ')' DOPPELDOPPEL f_arg_list
              | IDID f_arg_list
              ;
f_arg_list    : f_arg_list ',' f_arg
              | f_arg
              ;
intent        : IN
              | OUT
              | INOUT
              ;
f_arg         : IDID
              | IDPAR '(' numlist ')'
              | arg
              ;

f_function    : SUBROUTINE fctname '(' f_parameters ')' fortran
                  para_def_io
                 statements
                END SUBROUTINE fctname
              | FUNCTION fctname   '(' f_parameters ')' fortran
                  para_def
                  statements
                END FUNCTION fctname
              ;
fctname       : ID
              | IDPAR
              | IDID
              ;
statements    : statements statement
              | statement
              ;
bloblist      : bloblist ',' ID
              | bloblist ',' IDCOM
              | ID
              | IDCOM
              ;
statement     : skip
              | assignment
              | proccall
              | ifstatement
              | loop
              ;
skip          : SKIP
              | SKIP ';'
```

```
              ;
assignment    : array EQ cond
              | array EQ cond ';'
              | array ASSIGN cond ';'
              ;
array         : ID
              | IDPAR '[' exprlist ']'
              | IDPAR '(' exprlist ')'
              ;
proccall      : CALL procedure
              | procedure ';'
              | procedure
              ;
procedure     : IDPAR '(' exprlist ')'
              | IDPAR '(' ')'
              ;
ifstatement   : C_IF save_stmnt '(' cond ')'
                 c_body %prec C_ELSE
              | C_IF save_stmnt '(' cond ')'
                 c_body C_ELSE c_body
              | IF save_stmnt '(' cond ')' THEN statements END IF
              | IF save_stmnt '(' cond ')' THEN statements ELSE statements END IF
              ;
save_stmnt    :
              ;
c_body        : '{' statements '}'
              | '{' '}'
              | statement
              ;
index         : ID
              ;
index_        : ID
              ;
op            : LE
              | LEQ
              | GE
              | GEQ
              | EQ
              | EQEQ
              ;
loop          : c_loop
              | f_loop
              | s1_loop
              | s2_loop
              ;
c_loop        : FOR '(' index EQ expr ';' index_ op expr ';' c_step ')' level
                c_body
              | C_WHILE '('  cond ')' level c_body
              ;
c_step        : index_ PLUSPLUS
              | index_ MINUSMINUS
              | index_ EQ index_ PLUS  NUM
              | index_ EQ index_ MINUS NUM
              ;
f_loop        : DO funArray index EQ expr ',' expr f_step level f_s1_body END DO
              | DO funArray WHILE '('  cond ')' level f_s1_body END DO
              | DO funArray index EQ expr WHILE '(' cond ')' level f_s1_body END DO
              ;
level         :
              ;
funArray      :
              ;
f_step        : ',' sign NUM
              |
              ;
```

```
sign          : PLUS
              | MINUS
              |
              ;
f_s1_body     : statements
              |
              ;
s2_loop       : FOR funArray index EQ expr LEFT expr s2_step level statements
              ;
s2_step       : RIGHT sign NUM
              |
              ;
s1_loop       : FOR funArray index ASSIGN expr TO expr s1_step DOs level f_s1_body OD
              ;
s1_step       : STEP sign NUM
              |
              ;

exprlist      : exprlist ',' expr
              | expr
              ;
expr          : '(' expr ')'
              | expr PLUS  expr
              | expr MINUS expr
              | expr TIMES expr
              | expr  DIV  expr
              | expr  MOD  expr
              | PLUS  expr
              | MINUS expr %prec UMINUS
              | array_index_fctcall
              | NUM
              ;
array_index_fctcall
              : IDPAR '[' exprlist ']'
              | IDPAR '(' exprlist ')'
              | IDPAR '(' ')'
              | IDCOM
              | IDID
              | ID
              ;

dep           : dep DEP NUM NUM
                matrix NUM NUM
              | DEP NUM NUM
                matrix NUM NUM
              ;
matrix        :
              ;
cond          : '(' cond ')'
              | expr
              | cond OR  cond
              | cond AND cond
              | NOT cond %prec UMINUS
              | compare
              ;
compare       : expr op expr
              ;
```

The terminals are defined in the file <*LooPoDir*>/scanpars2/scan.l:

```
ws         [ \t]+
number     [0-9]+
ident      [a-zA-Z][_0-9a-zA-Z]*
```

```
%s comment

%%
"\n"             { lineno++; }
"/*"             { BEGIN(comment); }
<comment>"*/"    { BEGIN(0); }
<comment>[^*\n]+ ;
<comment>.       ;
{ws}             ;
{number}  { yylval->value=atoi(yytext);
            return loopoParser::NUM ; }
for       { return loopoParser::FOR ; }
to        { return loopoParser::TO ; }
step      { return loopoParser::STEP ; }
do        { return loopoParser::DOs ; }
od        { return loopoParser::OD ; }
CONSTANT  { return loopoParser::CONSTANT; }
while     { return loopoParser::C_WHILE ; }
WHILE     { return loopoParser::WHILE ; }
if        { return loopoParser::C_IF ; }
else      { return loopoParser::C_ELSE ; }
return    { return loopoParser::RETURN ; }
IF        { return loopoParser::IF ; }
THEN      { return loopoParser::THEN ; }
ELSE      { return loopoParser::ELSE ; }
IN        { return loopoParser::IN ; }
OUT       { return loopoParser::OUT ; }
INOUT     { return loopoParser::INOUT ; }
INTENT    { return loopoParser::INTENT ; }
SUBROUTINE { return loopoParser::SUBROUTINE ; }
FUNCTION  { return loopoParser::FUNCTION ; }
DO        { return loopoParser::DO ; }
END       { return loopoParser::END ; }
CALL      { return loopoParser::CALL ; }
dep       { return loopoParser::DEP ; }
denom     { return loopoParser::DENOM; }
skip      { return loopoParser::SKIP ; }
void      { return loopoParser::NOTYPE ; }
AND       { return loopoParser::AND ; }
OR        { return loopoParser::OR ; }
NOT       { return loopoParser::NOT ; }
"||"      { return loopoParser::OR ; }
"&&"      { return loopoParser::AND ; }
"!"       { return loopoParser::NOT ; }
":="      { return loopoParser::ASSIGN ; }
"::"      { return loopoParser::DOPPELDOPPEL ; }
"<"       { return loopoParser::LE ; }
"<="      { return loopoParser::LEQ ; }
"="       { return loopoParser::EQ ; }
"=="      { return loopoParser::EQEQ ; }
">="      { return loopoParser::GEQ ; }
">"       { return loopoParser::GE ; }
"<-"      { return loopoParser::LEFT ; }
"->"      { return loopoParser::RIGHT ; }
"++"      { return loopoParser::PLUSPLUS; }
"--"      { return loopoParser::MINUSMINUS ; }
"+"       { return loopoParser::PLUS ; }
"-"       { return loopoParser::MINUS ; }
"*"       { return loopoParser::TIMES ; }
"/"       { return loopoParser::DIV ; }
"%"       { return loopoParser::MOD ; }
```

References

[Ban93] Utpal Banerjee. *Loop Transformations for Restructering Compilers: The Foundations.* Kluwer Academic Publishers, 1993.

[DR94] Alain Darte and Yves Robert. Constructive methods for scheduling uniform loop nests. *IEEE Trans. on Parallel and Distributed Systems*, 5(8), August 1994.

[DV94] Alain Darte and Frédéric Vivien. Automatic parallelization based on multi-dimensional scheduling. Technical Report 94-24, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, September 1994.

[Ell95] Nils Ellmenreich. Das Frontend von LooPo. Unpublished Report, October 1995. http://www.uni-passau.de/~loopo/doc/ellmenreich-p.ps.Z.

[Fab95] Peter Faber. LooPo target-output. Unpublished Report, February 1995. http://www.uni-passau.de/~loopo/doc/faber-p.ps.Z.

[Fea88] P. Feautrier. Parametric integer programming. *Operations Research*, 22(3):243–268, 1988.

[Fea91] P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, February 1991.

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *Int. J. Parallel Programming*, 21(5):313–348, October 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Programming*, 21(6):389–420, October 1992.

[Fea94] Paul Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.

[Gün96] Robert Günz. The new LooPo scanner and parser. Unpublished Report, February 1996. http://www.uni-passau.de/~loopo/doc/guenz-p.ps.Z.

[Kei96] Harald Keimer. Vergleich von Verfahren zur Datenabhängigkeitsanalyse. Master's thesis, Universität Passau, 1996. In preparation.

[KMW67] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[KPR94] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. Technical Report CS-TR-3317, Dept. of Computer Science, Univ. of Maryland, 1994.

[Lam74] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[Len93] Christian Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[LG95] Christian Lengauer and Martin Griebl. On the parallelization of loop nests containing while loops. In N. N. Mirenkov, Q.-P. Gu, S. Peng, and S. Sedukhin, editors, *Proc. 1st Aizu Int. Symp. on Parallel Algorithm/Architecture Synthesis (pAs'95)*, pages 10–18. IEEE Computer Society Press, 1995.

[Mei96] Wolfgang Meisl. Practical methods for scheduling and allocation in the polytope model. Master's thesis, Universität Passau, 1996. In preparation.

[Par93] Parsytec Computer GmbH. *PARIX 1.2 Software Documentation*, March 1993.

[Rao85] S. K. Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays.* PhD thesis, Department of Electrical Engineering, Stanford University, October 1985.

[Sch95] Frank Schuler. The LooPo scanner and parser. Unpublished Report, January 1995. http://www.uni-passau.de/~loopo/doc/schuler-p.ps.Z.

[Wet95] Sabine Wetzel. Automatic code generation in the polytope model. Master's thesis, Universität Passau, November 1995. http://www.uni-passau.de/~loopo/doc/wetzel-d.ps.Z.

[Wüs95] Alexander Wüst. Dispo2 - Graphische Darstellung der Indexräume verschachtelter Schleifen. Unpublished Report, November 1995. http://www.uni-passau.de/~loopo/doc/wuest-p.ps.Z.