An Expert Advisor for Parallel Programming Environments and Its Realization within the Framework of the Vienna Fortran Compilation System *

Stefan Andel **Barbara M. Chapman** **Jan Hulman**
Hans P. Zima

Institute for Software Technology and Parallel Systems
University of Vienna, Liechtensteinstr. 22,
A-1090 VIENNA, AUSTRIA

E-Mail: {andel,barbara,hulman,zima}@par.univie.ac.at

Abstract

A large collection of useful, though usually fragmented, knowledge has been acquired by the experts in program parallelization and optimization. It is highly desirable to offer this expert knowledge to the average user and therefore incorporate it into the new generation of parallelizers. The contemporary parallel compiler technology does not allow straightforward embedding of expert knowledge into parallelizers but the application of the present-day expert system technology establishes the platform for combination of the precise analytical and numerical information from the transformation system and performance prediction tools with the expert knowledge about parallelization. Moreover, this technology supports the rapid prototyping and expandability, it forms the logical integration platform for the separate parts of the parallel programming environment, and it supports the creation of intelligent dialog with the user during the parallelization session.

This material presents the basic ideas underlying the design and implementation of an expert advisor which yields an expert advice to the programmer during the program parallelization and optimization process.

A knowledge based tool, eXPert Advisor (XPA), has been proposed to support the parallelization process under the Vienna Fortran Compilation System (VFCS). The basic components of this tool, as it has been implemented, are described. The interconnection of XPA with VFCS, program model, structure of XPA's knowledge base, and knowledge application are outlined. An example of the XPA application in the field of the automatic data distribution is presented. Finally, we outline our future research directions in the field of knowledge-based parallelization.

*The work described in this paper was partially supported by the Austrian Research Foundation (FWF Grant P9205-PHY) and was performed within the ESPRIT research project PPPE (P6643).

Contents

1 Introduction

Modern distributed-memory multiprocessor systems (DMMPs), such as those offered by Meiko, Intel, Thinking Machines and NCUBE, are able to provide their users with a sustained high performance at comparatively reasonable cost. Recent systems are scalable to large numbers of processors; some systems, (e.g. Meiko's CS-2) offer both scalar and vector processing nodes.

It is, however, very difficult for a programmer to utilize the power of these machines. Established programming paradigms based on message-passing require the user to distribute both data and work explicitly among processors. This task is often likened to assembly language programming on sequential computers. The programming difficulties are further increased when the nodes of a system are no longer homogeneous. As a result, application development on DMMPs is a laborious, time-consuming and error prone process.

A long-term goal of language and compiler research is to make program development on DMMPs as productive as programming on conventional sequential architectures. One approach to this task is to provide appropriate extensions to existing programming languages, so that the user can write programs with a single thread of control and global data references only. It is then the task of system software to convert the user program into a form suitable for execution on a DMMP; we refer to this process as *parallelization*. This programming paradigm promises not only faster and easier program development, but also greater flexibility and portability of codes.

The Vienna Fortran Compilation System (VFCS) is a state-of-the-art parallelizing compiler. However, the parallelization of a program within VFCS still remains a complex optimization process whereby the user is forced to drive the parallelization session. The most crucial and critical decisions which he must take are related to the specification of a distribution of program arrays and to the selection and application of appropriate program optimizing transformations. These decisions require detailed knowledge about the program, compiler features, and target machine characteristics. The user makes decisions on the basis of previous experience (heuristics), and by exploitation of information from available performance analysis tools. Therefore, decision-making on the basis of a large spectrum of information, ranging from detailed analytical information, via performance parameters, up to human heuristics and known ad-hoc solutions, is required.

An average user usually does not possess enough knowledge to take such decisions; a knowledge gap exists between him and the system. With increasing complexity of the parallelization environment and variety of target architectures, the gap is too wide for users to achieve a peak parallel program performance, unless they are guided by an expert advice.

We claim that the representation and exploitation of *explicit* expert knowledge covering various parallelization aspects within the current parallelizers can enable their future evolution and will make them more automatic and intelligent.

The contemporary *expert system technology* is applicable to organize and represent a large amount of knowledge and information from different sources. The application of expert system technology in the field of parallelization leads to a new dimension: it establishes a platform for the combination of precise analytical and numerical information yielded by the compilation system and performance prediction tools, with human knowledge about parallelization, which is often imprecise and heuristic in its nature.

The expert system methodology, based upon explicit representation and exploitation of the knowledge related to a particular problem domain, has proved to be suitable and fruitful in a number of similar problem areas (particularly CAD/CAM) [15].

In this paper we describe the design of an environment for parallelization which uses expert system technology in an attempt to improve the state of the art. Much of the discussion focuses on a specific component of this, a new tool currently being implemented at the University of Vienna. This tool, the eXPert Advisor (hereafter referred to as **XPA**), is being built using the expert system methodology. XPA cooperates with the other tools in this environment in an attempt to both make the parallelization process more automatic and convenient, as well as to improve the quality of the code generated.

This paper is organized as follows. In Section 2 we give an overview of the VFCS parallelization environment, and discuss the parallelization process within such environment. In Section 3 we describe different types of expert advice for supporting a parallelization session in general, and Section 4 elaborates this topic in more detail with specific focus on the VFCS parallelization environment. In Section 5 we then outline the structure and main modules of the Expert Advisor.

The rest of the paper is devoted to the XPA prototype as it is currently implemented, its implementation requirements and implementation substrate, and its integration into VFCS parallelization environment. We give a detailed description of the XPA program model and knowledge base rules. This includes an outline of the approach it takes to derive data distribution for arrays in the source program. The XPA session is described and several snapshots are presented to illustrate the way XPA works. We conclude with a discussion of future research in knowledge-based parallel program development and give some developments of Expert Advisor as well as an overview of related work.

2 Features of VFCS Parallelization Environment and Parallelization Process

In this section we describe the VFCS parallelization environment by presenting it's components (see Figure 1) and by describing their capabilities. The program parallelization and optimization process, as it is viewed by the programmer, will be discussed.

Parallelization environment is centered around the **Vienna Fortran Compiler (VFCS)** [8], a *source-to-source translator* from **Vienna Fortran** [33] and **HPF** [17] into **Message Passing Fortran** (MPF). The current system can generate parallel programs for several target architectures (**Meiko CS-2**, **Intel iPSC 860**, **Intel Paragon**, **Convex Exemplar**) using **MPI** and **NX** communication libraries, as well as providing an additional independent target in the form of the **Parmacs** routines.

VFCS can be used as a command line compiler for Vienna Fortran and HPF programs, in which case it applies a standard set of program transformations and optimizations to the code, or it can be used as an interactive parallelization system where the user has full control of the parallelization process.

However, many of the decisions which have to be made during interactive parallelization driven by the user must be supported by knowledge of the program's behavior at run time which cannot be derived from the program by analytical methods alone. To support the derivation of such information a set of performance prediction and analysis tools is incorporated into the parallelization environment. These tools are able to cooperate with VFCS, and may be invoked from within it. As we shall see, they will also cooperate with XPA, for which they represent a further source of information which can form a base for activation of various parts of knowledge present in the

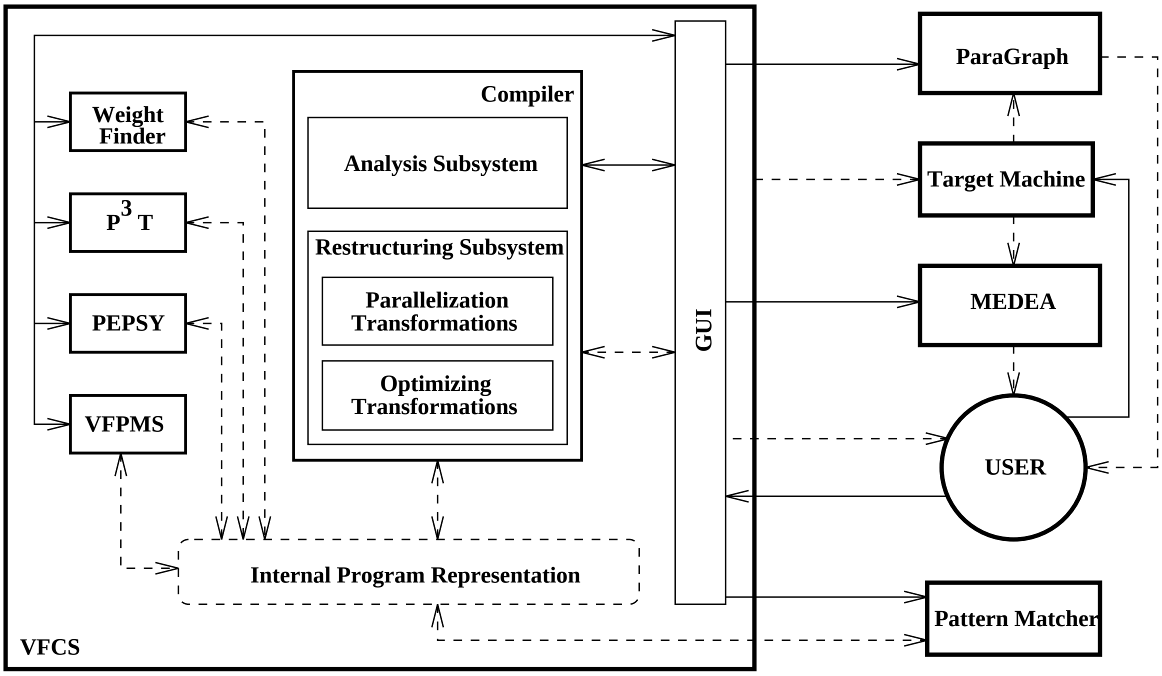

Figure 1: VFCS Parallelization Environment for Regular Codes

knowledge base.

Before we describe the integrated tools of the VFCS parallelization environment, we briefly discuss the VFCS parallelization model.

2.1 The Programming Model

During VFCS program parallelization, target code is generated according to the **SPMD** (Single Program Multiple Data) model of computation, in which the arrays of the original program are partitioned and mapped to the target processors (*data distribution*). The data owned by a processor according to the distribution are stored in its local memory. Now the work is distributed according to the data distribution: computations which define the data elements owned by a processor are performed by it (this is known as the *owner computes* paradigm). Essentially the same parameterized program is then executed on all processors computing the values for different partitions of the arrays. If accesses to non-local data occur in the code, the compiler automatically inserts communication statements to send and receive data at the appropriate positions in the code (this is called *message passing*). The compiler will also map logical processor structures declared by the user to the physical processors which execute the program. These transformations are, however, transparent to the HPF or Vienna Fortran programmer.

2.2 Tools Incorporated in VFCS Parallelization Environment

2.2.1 The Compiler

All the modules of the compiler are centered around an internal program database. This internal program representation is created at the loading of an input program into VFCS. All compiler components work with and modify this internal representation. Only during the final phase of compilation, the code generation, is this internal program representation translated into textual form (target machine program).

The translation from Vienna Fortran to Message Passing Fortran can be described as a sequence of the following five phases:

During the first phase (**Front End**) the source code is analyzed, normalized, and translated into an *internal representation* suitable for further compiler manipulation. Besides the syntax tree, flow graph, call graph, and symbol table, the internal representation contains precise dependence information resulting from exhaustive data flow and data dependence analysis. Further, some standard transformations such as constant propagation, dead code elimination and expression simplification may be applied at this stage.

The next phase (**I/O splitting**) splits the normalized program into a *host program* and a *node program*. The host program will, after compilation, be executed on the host computer or a dedicated node of the target system as a *host process*. It performs global management tasks, such as allocation of resources, node programs loading, execution and terminating; it also handles all input/output operations. The node program, still sequential, contains the actual computation. In its body, all I/O statements are replaced respectively by receive/send statements to the host process.

The third phase (**initial parallelization**) transforms the data distribution specification within the node program into *mask* and *communication annotations*. A boolean mask is associated with each statement of the node program to fulfill the owner computes paradigm. For all non-local data accesses, *communication insertion* generates communication statements which copy all used non-local data items to private variables of the processor. Note that the performance of code to which only these transformations are applied is likely to be very poor.

The fourth phase (**optimization**) transforms program into an *optimized parallel MPF* program. Communication statements and masking are improved: communication statements are moved out of loops, aggregated, and strip mining of loops across the processors is performed by propagating the mask semantics to local loop bounds, whenever possible. Performing of *overlap analysis* is a precondition for such optimization together with precise flow and data dependence analysis at the first phase.

The last phase (**Back End**) optimizes the parallel program for the selected target architecture and generates the *target code*.

At this place, we note only that program parallelization and parallel program optimization using VFCS is an interactive and iterative process that involves, as a rule, intensive interactions with a user[1].

[1]The VFCS may be used also in a batch mode without user interaction. In that case, a predefined sequence of parallelization actions is performed.

2.2.2 Transformation Subsystem for Program Optimization

Program parallelization within VFCS, as described in Section 2.2.1, can be seen as a program restructuring process. The separate phases are realized as applications of the appropriate implicit restructuring transformations. Besides those implicit transformations, a catalog of explicit restructuring transformations is included in VFCS. These transformations can be applied to the parallelized program under the user's control.

The available explicit restructuring transformations can be divided according to their functionality into two groups:

Transformations improving program analysis. Transformations of this group do not have significant impact on the target program performance. In general, they are applied during earlier phases of program parallelization (during and immediately after frontend) with the aim of substantially improve the precision of program analysis (in particular data dependence analysis for regular array references inside the program). This group comprises: if-conversion, interprocedural constant propagation, scalar forward substitution, induction variable substitution, expression simplification, and procedure inlining [34]. Although there is not one optimal order for applying these to a program, the transformations to propagate constants, and to improve information on array subscripts should be performed first.

Transformations improving program performance. This group comprises transformations such as: loop distribution, loop interchange, loop unrolling, strip mining, vector code generation, and simple kernel reductions. As opposite to the previous group, an application of these transformations can significantly influence the performance of the target parallel program. Whether these transformations are to be applied, and to which parts of the code, requires a great deal of information about the program structure and about their effects on the target program performance. As a rule, a decision must also take the features of the target architecture such as interconnection topology, communication system parameters, size of the cash memory, number of processors, as well as the specified data distribution, into account.

2.2.3 The Pattern Matcher for VFCS

In [12] the implementation of a prototype tool that performs recognition of algorithmic patterns, integrated within the VFCS, has been described. It starts from a *base internal representation* of the program, derived from the structural analysis performed by the VFCS front-end, and builds a hierarchical description of the program in terms of concept instances, until one or more parallelizable algorithmic patterns are recognized. The tool has been implemented in Prolog: this choice has been mainly driven by its capability to be utilized as a deductive inference-rule engine to perform the hierarchical concept parsing.

The use of a rule-based system as Prolog, and the specification of (inter-statement level) patterns by control and data dependence relations, permits the successful handling of a number of problems arising in the context of *program comprehension*, that can be summarized as follows:

- *syntactic variation*;
- *delocalization*: the implementation of a pattern can be spread throughout the code;

- *implementation variation*: an abstract concept can be implemented in many different ways, and so has to be represented by several and alternative patterns;
- *overlapping implementations*: often the implementations of two or more distinct concepts are merged, so portions of a program may be part of more than one pattern.

The prototype is operational, and able to recognize a few dozen concepts related to numerical computation.

It provides two types of output: first, a graphical browser that permits visualization of the hierarchical description of the recognized patterns, in the form of a graph, and their implementation within the program code: the user may select the nodes in the graph that represent subconcepts, and the corresponding code fragments implementing them are highlighted. The second, the attribution of nodes in the syntax tree is performed.

2.2.4 Performance Prediction and Analysis Tools

Each elementary parallelization step such as selection of a restructuring transformation or the distribution of a particular array should be evaluated with regard to its performance impact on the target parallel program. In rare situations, the performance impact can be partially estimated in an analytical way. However, for more complex cases a tool from the rich set of VFCS performance analysis tools must be used to obtain appropriate performance parameters.

The **VFCS performance prediction and analysis subsystem** comprises several classes of performance tools: the *static tools* performing analysis based only upon statically derivable information (P^3T) [14], the *simulation tools* executing an abstract version of the program (PEPSY) [4], and the *post-execution performance analysis tools* that require the sequential or parallel program run, trace the program execution, and then postprocess the trace files generated; the statistical methods and *visualization toolkits* may be applied (Weight Finder, VFPMS, MEDEA, ParaGraph) [13, 32, 26, 7, 29].

Each of the performance tools which will be mentioned below computes the particular set of *performance indices*. The indices sets can mutually overlap, however the achieved precision of the particular indices and the cost of their computation may significantly differ among tools.

The invocation and exploitation of the particular performance tool is highly dependent on the actual parallelization subproblem being solved and it involves several aspects: specification of the program part to be analyzed, performance indices to be computed, selection of the performance tool capable of returning indices of required precision, and interpretation of performance results. Moreover, some performance indices are presented in a graphical form that is difficult to be interpreted automatically, so the user is expected to serve as a link between the performance tool and a parallelizer or a parallelization support tool.

Some of the performance tools in the VFCS parallelization environment are separate tasks, only loosely coupled with the compiler, in contrast to the other environment components that are directly integrated (linked) with the compiler routines.

The Weight Finder is an advanced sequential profiler of Fortran 77 programs. It provides the following features:

- localize computational intensive program parts by static/or dynamic performance measurements of program parts (e.g. DO-loops, procedure calls, entire program)
- derive frequency values for program parts (DO-loops, procedure calls, assignment statements, entire program)
- average values for varying DO-loops parameters
- computes true ratios for conditional statements

P^{3}T is an interactive performance estimator for parallel programs that assists the user in performance tuning of scientific Fortran programs. Its main purpose is to detect performance bottlenecks in the program before its execution on the distributed memory parallel architecture. Four of the most critical performance aspects of parallel programs are estimated: *load balance*, *cache locality*, *communication* and *computation overhead*. P^{3}T computes a set of parallel program parameters each of which reflects a different performance aspect. In the following all parallel program parameters are briefly outlined. A detailed description can be found in [14].

- *Work Distribution*: defines how the work contained in a loop nest, a procedure call, or in the overall program is distributed across the processors executing the program
- *Number of Transfers*: estimates the number of send/receive operations induced by a communication statement
- *Amount of Data Transferred*: approximates the data volume transferred over the communication network by a communication statement
- *Network Contention*: represents an upper bound of the number of channel contentions induced by a communication statement on the underlying communication network
- *Transfer Time*: predicts the time needed to transfer the required non-local data over the physical communication network
- *Number of Cache Misses*: models the data locality of data accesses; it is defined by an upper bound of the number of accessed cache lines, which is assumed to correlate with the number of cache misses
- *Computation Time*: estimates the computation time of a loop nest, a procedure call, or the whole program

P^{3}T is based on a single sequential profile run to provide concrete data for statement and loop execution counts and branching probabilities. Hence, all parameters as estimated by P^{3}T are strongly based on the values from the sequential profiler.

Without an intelligent support tool (as an expert advisor) integrated in the parallelization environment, it is the user's responsibility to interpret the obtained P^{3}T parameters and to decide what code changes to apply in order to improve the corresponding performance outcome.

PEPSY is a simulation tool for automatic modeling and evaluation of detailed and global performance characteristics of parallel programs modeled in an abstract form. It computes *static* and *dynamic* performance parameters.

Static parameters characterize the size and structure of the parallel program. Dynamic parameters characterize the behavior of the parallel program for steady and transient states. PEPSY evaluates about ten basic and several derived dynamic indices (e.g. execution time, computation time, communication time, parallelism degree, utilization, communication volume, etc. Comparing different versions of the parallel program, it evaluates the *scalability* parameters, e.g. execution signature, speed-up, efficiency, efficacy. Monitoring facilities of PEPSY evaluate the performance indices for the whole execution period of the parallel program or for the detailed dynamic behavior, evaluating the values by an optional sampling period.

The performance results are provided for the global level, as an average values for all processors, for the processor level comparing all parallel processes, e.g. for the load balancing analysis, and for the intraprocess level, i.e., the statement level of the parallel program.

Vienna Fortran Performance Measuring System (**VFPMS**) supports the semi-automatic measurements of Vienna Fortran and HPF programs. It is implemented as a fully integrated module of the Vienna Fortran Compilation System.

The system allows the user to specify measurements of selected program components (subroutines, loops, ranges, communications) either during an interactive VFCS session or by using VFCS batch measurement commands. VFPMS automatically computes the required measuring points and generates an instrumented MPF node program as well as a detailed measuring description. Running the instrumented program on a parallel architecture produces a measurement file which can be used for post execution performance analysis and monitoring.

MEDEA is a general-purpose environment for the analysis of performance data collected by measuring/monitoring tools. Various analysis techniques for the characterization of parallel program are integrated in it.

The static property of the performance data set can be examined through the multidimensional *clustering analysis* techniques to identify groups of elements (units of work) having homogeneous characteristics with respect to predefined parameters.

Besides the quantitative description of the program obtained by statistical analysis, the *functional analysis* gives its "logical" description. This approach allows one to identify, for example, the membership of particular components in a specific cluster.

The application of numerical *fitting* permits reproduction of the time-varying characteristics of the programs. Moreover, the fitting analysis provided by MEDEA allows the user to derive, from the measured data, an analytic description of the dynamic aspects of the program described by means of one or more collected parameters.

Besides other, the MEDEA allows the computation of *scalability indices* such as *speedup*, which gives the performance gain of parallel processing versus sequential processing, the *efficiency* that represents the speedup per processor and the *efficacy* which indicates how well the processors are used.

Finally, a graphical visualization of the parameter values and of the results produced by each analysis performed is provided.

ParaGraph is a "post-processing" graphical display system for visualizing the behavior and performance of parallel programs. It provides a visual replay of the events that actually occurred when a parallel program was executed on target architecture. The tool provides several displays (e.g. utilization displays, communication displays, etc.).

2.3 Parallelization within VFCS Environment

In this section we outline the most important problems which the user has to overcome during the parallelization and optimization of a program in the interactive VFCS parallelization environment. After this discussion the decomposition of the task of the program parallelization and optimization will be presented.

2.3.1 Description of the User Driven Parallelization within VFCS

The specification of the data distribution is the crucial and most critical step in the parallelization process, requiring a detailed knowledge of the algorithm, the compiler, the architecture, and the cost of communication generated in the transformed program. The penalty for a "wrong" decision may be quite high: excessive communication may be generated, resulting in an unacceptable performance degradation of the target program. Thus the user's skill in selecting data distributions in a Vienna Fortran program may have a large influence on the quality of the code produced.

The main task of a VFCS programmer is thus to **specify distribution of arrays** in the code. This requires declaring the set of processors with a suitable logical structure, specifying the way the arrays have to be distributed to processors, and deciding how the arrays have to be distributed relative one to other. The way the data is distributed determines the memory allocation, the type and amount of communication, and the workload on each processor; hence, the data distribution also determines the overall performance of the parallelized program. Program factors that influence the selection of a distribution include the size of the application, the ranges of do-loops, major access patterns of arrays, how the arrays are used together, and the dependences among the statements of a loop. The communication behavior of the target machine, its communication/computation ratio, cache sizes and pipelines/vector processing facilities all play a role.

The other task of the VFCS user is the **selection of appropriate program optimization transformations** and the sequence of their application. The preconditions of every transformation application, the optimization goal, and the effect of an application must be known to the user. This task is difficult and it may seriously influence the performance of the resultant parallel program.

The **control of a parallelization session** in a parallelization environment, such as VFCS, puts further demands on the VFCS programmer. He must possess the knowledge *when* to use the tools integrated into the VFCS environment, *how* to use these tools, and he must be able to *understand* the information obtained from these tools. **Decision making** on the basis of a large spectrum of information, often of different nature, is required.

2.3.2 Decomposition of the Parallelization within VFCS

The decomposition of a difficult problem into simpler ones is one of the most powerful methods of human thinking. The application of this principle at the construction of an expert advisor for program parallelization aims to divide a parallelization problem into many subproblems. Specialized knowledge dedicated for solution of every subproblem can be formulated. According to the decomposition of the parallelization into subproblems, the corresponding hierarchical structure of the knowledge base can be constructed. Therefore, *fragmented* knowledge of different abstraction levels can be incorporated into the expert advisor knowledge base and organized in the natural way. Moreover, the modular structure of the knowledge base imposed by problem decomposition positively influences the expert advisors expandibility and flexibility, and improves reuse of knowledge.

Due to the complexity of the parallelization problem, it is not possible to find an "optimal solution" (parallel program with acceptable performance) at once, but usually the original, often weak, initial proposal can be constructed and then gradually refined to reach the resultant parallel program. This refinement phase, often called **program tuning**, is obviously of an **iterative nature**. The program tuning may comprise several iterations over the whole parallelization process to obtain a parallel program with acceptable performance.

In the following, the decomposition of parallelization within VFCS into subproblems and program tuning phases is outlined. As can be seen in Figure 2, the input sequential or (partially) annotated program is gradually transformed by the VFCS (initial parallelization, communication optimization, mask optimization, backend) to obtain a parallel program for the target architecture. This program is the source for the target compiler. After its compilation, the program can be executed on the parallel machine. In the figure, the boxes represent different transformation stages of the parallelized program). On the left-hand side of every box, the main parallelization activities performed by the user (in corresponding program stage) are presented. On the right-hand side there are the tools supporting these user activities.

From the point of view of automation of the parallelization process, it is important to decompose the parallelization into **tuning phases** whereby each phase corresponds to one iteration subcycle of program tuning. In every phase, one or more specific parallelization problems are solved or re-solved.

In the first phase, the main problems are a specification of the data distribution and an appropriate application of optimizing transformations. The Weight Finder and the Pattern Matcher may be invoked to obtain supporting information. Other information, e.g. about data dependences, is supplied by the compiler's program analysis module. In this phase the user is responsible for evaluating the impact of data distribution and applied transformations on the performance of the parallel program which would be generated from the current annotated one (the output of this phase). No support tools for such an evaluation are available in the current VFCS parallelization environment.

In the second phase, the annotated program is transformed to a message passing one. The communication and mask optimization transformations are realized. Now the user obtains more information related to the parallel program. He/she may see where communication statements have been inserted into the program. From the overlap information corresponding to each communication statement(s) he may deduce whether the statement causes an extensive communication or not. Other information influencing evaluation of the current program may be obtained by an investigation of mask expressions in program statements. But all this information is not quantitative, moreover it is local to a particular communication, assignment or DO statement. Appropriate conclusions related to the modification of data distribution can be formulated. The P^3T can be used to support these decisions by more precise information [14]. As mentioned in Section 2.2.4, it supplies a set of parameters, which predict particular performance characteristics of the program. After successful interpretation of these parameters, appropriate decisions to modify data distribution or

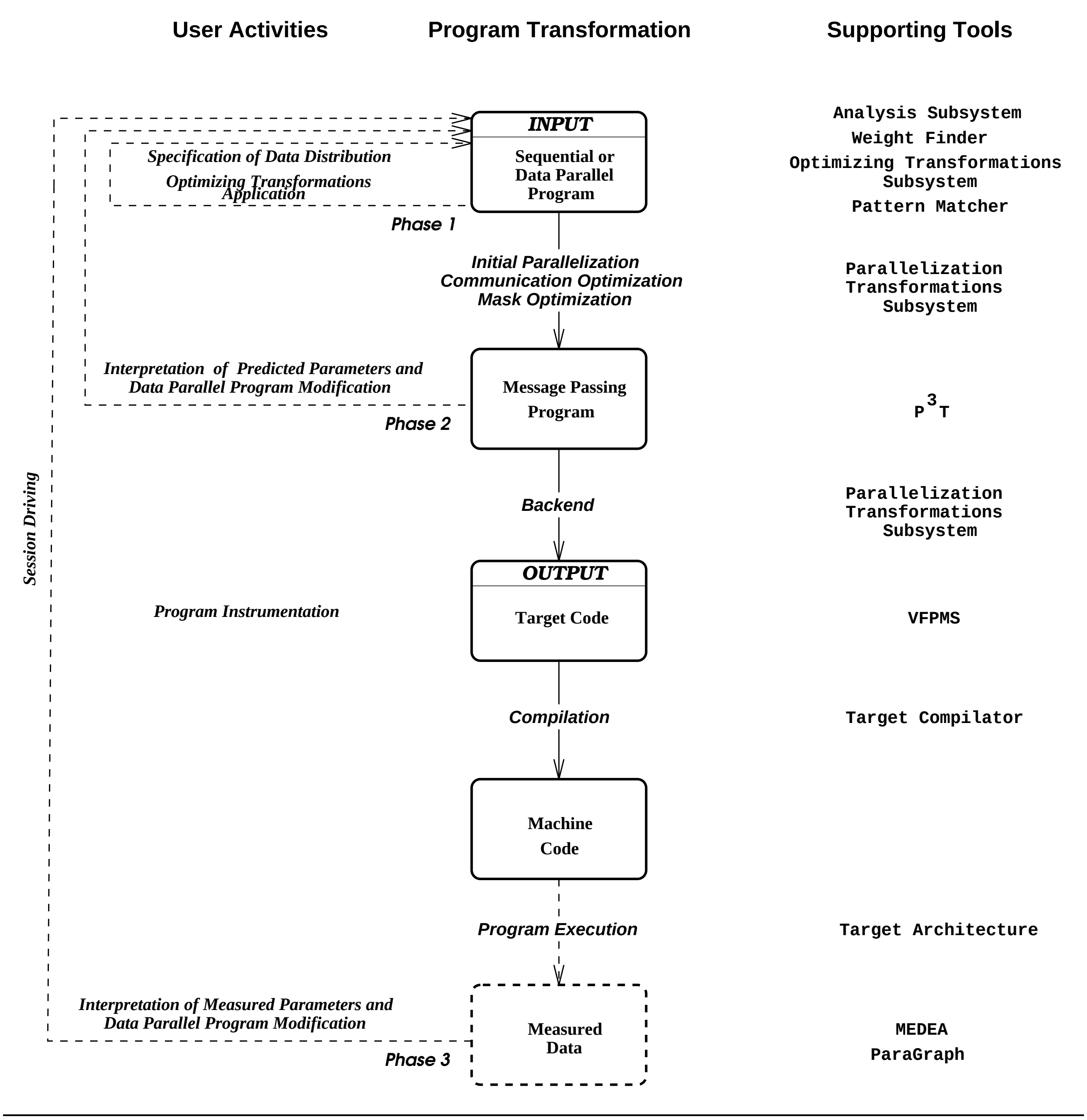

Figure 2: Parallelization Session within VFCS Environment

to apply other transformations can be made.

At the beginning of the third phase, the program is transformed by the backend of the VFCS compiler into the target code. This code can be an output from the parallelization process and may serve as a source to the target compiler. But it is obvious that the other steps in this tuning phase will be often carried out. To obtain the most precise information about the performance of the parallel program, the code must be instrumented (with VFPMS), the target compilation performed, and the program be executed on the target architecture. The user is responsible for selecting which parameters to measure and where in the program they are to be measured. The performance analysis tools are used to analyse the measured data, often large sets. The interpretation of results of this analysis should influence the user's decision on where/how to modify the program data distribution and optimizing transformations.

The above mentioned tuning phases may be performed many times before an acceptable parallel program is obtained.

3 Three Variants of Expert Advice

To produce an executable parallel program for a particular architecture, and to guarantee that a compiled program will run efficiently on this architecture, an integrated environment supporting program parallelization is required. As may be infered from the previous sections, such an environment must integrate a parallel programming paradigm, compiling methodology, a complete set of performance prediction and analysis tools, and other supporting tools (e.g. pattern matcher, automatic data distribution tool, etc.). As a result, the programmer can not develop an efficient parallel program without understanding details of the program, the programming paradigm, the compiler, the hardware features of the target architecture, and the tools integrated in the parallelization environment. Rich parallelization experience is also inevitable to successfully parallelize a program in such environment. It is neither practical nor desirable to ask users to become experts in parallelization in order to be able to write efficient parallel codes. A better solution would be to supplement the parallelization environment with a new **intelligent tool**, integrating it in a unified parallel programming system. Moreover, this tool would offer parallelization expert knowledge to the novice or average user.

There are many methodologies for the development of such an intelligent tool. We aim to incorporate a tool which simulates a parallelization domain expert into the VFCS parallelization environment. In the following sections we suppose that the user is parallelizing a program and an expert in parallelization domain is ready to help him/her in particular situations occurring during the parallelization session. We distinguish three levels of expertise the expert can offer to the user. We outline them below and give example situations when they can be appropriately applied within VFCS parallelization. We are implementing these facilities in an expert advisor. They correspond to the three modes of expert advisor work. Of course, in particular situations, a combination of these three possibilities of advice need to be used.

3.1 Assistance

As a first type of expertise we consider **assistance**. Here, the expert *assists* the user to gain insight into understanding of different aspects of the parallelization and optimization problems. The user is active, he/she formulates questions, interprets the expert's answers and makes decisions based upon them related to particular parallelization subproblem. The expert is passive, he only answers the user's questions according to information available at the current parallelization state. Information, as presented by the expert, serves to improve orientation of the user within parallelization. The most valuable information is that is not explicitly visible in the program. The need to deduce and highlight it in the current situation arises from expert experience.

Another type of assistance, **advanced assistance**, could be appreciated by the user. It can be examplified by the situation when the user asks the expert what would happen, if he/she performs a particular parallelization step. The expert imagines (simulates in his mind) this new parallelization situation and gives exhaustive information related to the new potential parallelization state. On the basis of this information, the user can make an appropriate parallelization decision.

3.2 Guidance

As the second type of expertise we distinguish **guidance**. In a particular situation, which may happen within program parallelization, the expert *guides* the user to concentrate on a particular parallelization subproblem, recommends a particular parallelization action, gives intructions on how to realize this action, and explains the main reasons for performing it. It is clear that situations may occur, when the expert offers more than one possible way how to continue in parallelization, or is not able to recommend any particular parallelization step. When more than one recommendation is offered, the user is responsible for choosing an appropriate one.

3.3 Automation

Let us imagine a third variant of expertise. Here, the expert solves a particular parallelization subproblem and then gives the result to the user, or realizes the solution himself. The choose of the next parallelization step(s) can also be left to the expert and invoked by him **automatically**. In the first situation the user can not influence the search for the solution of a subproblem. In the second, the user is passive and the expert has temporary control of parallelization.

4 Expert Advice within VFCS parallelization

In this section we outline the application of expert advice variants for particular subproblems occurring during VFCS parallelization. We are implementing such advice within our expert advisor prototype. The communication between the user and expert advisor is realized by intelligent interfaces. That means, the user asks a question by choosing it from a a variety of menu driven user interfaces. The answers of the expert advisor are realized by special displays.

4.1 Assistance within VFCS

Exhaustive application of expert assistance (as can be seen later also of the other types of expert advice) must be integrated into the first phase of program parallelization (see Section 2.3.2), where two main problems, data distribution and application of optimizing transformations, have to be solved.

User interface for parallelizable components. First of all, the most important parts of the program (in our terminology *parallelizable components* or *program phases*) have to be identified. Identification of these program phases and rendering information about them is very important from the point of view of the human approach to parallelization, because not all information about a real program, often consisting of many thousands of lines, can be taken into consideration. To support this, the user interface for parallelizable components aims to assist the user in deciding which parts of the program to prefer when contradictory constraints on parallelization must be resolved. It displays the program phases together with percentage information of their sequential computation. The visualization of every individual program phase (as found by the expert advisor) is also supported. Percentage information attached to each program phase is computed from the time parameters obtained from Weight Finder sequential profiling.

User interface for arrays. To select a data distribution of the program arrays, important information about every array and relations among arrays is required. This is supported by the user interface for arrays. After selecting a particular array from it, the user can cross-examine e.g. align preferences for its dimensions. Moreover, he can inspect inter or intra dimensional align dimension preference conflicts of the array occurring within a unit, or in an interprocedural context. Highlighting of the contradictory align dimension preferences, which underlies every such align dimension preference conflict, is possible in the display window. The dimensions of the array which, when distributed, will cause sequential program execution are identified. Also the program phase where sequential program execution would take place can be highlighted. Data dependences which cause serialization of the program execution can be displayed, etc.

User interface for patterns. If the pattern matcher has been used, then the user can obtain information regarding the recognized patterns by the user interface for patterns. The statements of every recognized pattern can be highlighted in the source code display and parameters of the pattern, e.g. the array dimensions included in the pattern, are displayed. Using this interface the user can also ask the expert advisor to supply the optimal data distribution for arrays included in the pattern. This data distribution is of course only locally optimal. A knowledge cluster[2] dedicated for each recognizable pattern must be incorporated in the expert advisor knowledge base.

User interface for data distribution. An advanced assistance is supported by the user interface for data distribution. This intelligent and user friendly interface offers the possibility to specify the data distribution of program arrays in a convenient way. Arrays, together with their dimensions, are displayed in groups. Each group contains arrays which are declared in the same unit. The menu of acceptable distribution types is offered. The user selects an array, an array dimension and then he selects the data distribution type from the offered menu. Now he must specify any parameters of the data distribution (e.g. block sizes). After this, he can obtain the consequences of his decisions regarding the data distribution. First of all, information about the communication, which would be generated by the compiler, if the program were annotated by the proposed data distribution, is presented. Communication are related to the program phases where they would occur in the program, the communication patterns are identified, and the reason, why a particular communication statement is generated, is outlined. A more detailed description of

[2] We use the term *knowledge cluster* to denote the part of knowledge base which is related to a particular problem.

every predicted communication is given. Identification of program parts where mask optimization is not possible is also performed. Information giving program parts and their communication, mask optimization problems and parallelization degrees, can be generated.

The bulk of information shown in the above-mentioned user interfaces must be *deduced* from the information which is present in the actual state of parallelization. The details of deduction of new facts will be discussed later. An example of a simple rule from the knowledge base is presented in Figure 3. It serves the identification of inter-dimensional align preference conflicts.

4.2 Guidance within VFCS

Guidance for data distribution. The user interface for data distribution offers also a guidance regarding to *array privatization* and *array dimension replication.* The expert advisor gives recomendations relating to which arrays may be privatised or which dimensions of a particular array may be replicated to reduce communication and retain a high degree of parallelism. In an advanced version of our expert advisor, we could guide the user in deciding where in the program the *redistribution* of a particular array could be appropriate.

Guidance in finding parallelization strategies induced by interpretation of the P^3T parameters. The proper interpretation of performance estimates obtained from the P^3T at compile time, the support in finding effective strategies for performance tuning based on these estimates, are the key issues in the second phase (see Section 2.3.2) of the VFCS program parallelization. The expert advisor offers guidance in such program performance tuning. The bottlenecks in the program performance are identified, and the selection of effecient data distribution strategies and profitable program transformations is supported.

The top level guidance. Selection of the appropriate parallelization step(s) within VFCS parallelization, or procedure of using a particular support tool from the VFCS parallelization environment, can be a problem for a novice user. Therefore, the expert advisor must outline the scenario of the parallelization steps which seem to be the appropriate next steps in the current parallelization state. Providing instructions on how to use a tool from the parallelization environment could also be the task of expert guidance.

4.3 Automation within VFCS.

Automatic instrumentation of target code. Instrumentation of the target code, in order to obtain relevant parameters characterizing the program performance after its execution on the target architecture, is the key issue in the third phase of the parallel program performance tuning in the VFCS parallelization environment. In general, what should be measured and where the measurement should take place is specified by the expert advisor on the basis of the program model. Technically, the VFCS batch measuring commands are used to automatize instrumentation.

Automatic Data Distribution. As an alternative to the techniques presented in the previous sections regarding data distribution, an automatic solution of this problem can be also offered to the user. One possibility for realizing this alternative is to extend the parallelization environment by an external tool for automatic data distribution. The other possibility, that we prefer in our

Searching for Inter-Dimensional Alignment Conflicts

Rule Type: Forward Chaining

Rule Set: Alignment Conflicts

$\boxed{\textbf{IF}}$

- $AlignPref_1$ is align preference
- $ArrayDim_{11}$ of array $Array_{11}$ is in align preference with $ArrayDim_{12}$ of array $Array_{12}$ according to $AlignPref_1$
- $DimensionNumber(ArrayDim_{11}) == n_{11}$
- $DimensionNumber(ArrayDim_{12}) == n_{12}$
- $AlignPref_2$ is align preference
- $AlignPref_1 \mathrel{!=} AlignPref_2$
- $ArrayDim_{21}$ of array $Array_{21}$ is in align preference with $ArrayDim_{22}$ of array $Array_{22}$ according to $AlignPref_2$
- $DimensionNumber(ArrayDim_{21}) == n_{21}$
- $DimensionNumber(ArrayDim_{22}) == n_{22}$
- $Array_{11} == Array_{21}$
- $Array_{12} == Array_{22}$
- $n_{11} == n_{21}$
- $n_{12} \mathrel{!=} n_{22}$

$\boxed{\textbf{THEN}}$

- there is an **inter-dimensional alignment conflict** between dimensions of arrays $Array_{11}$ and $Array_{12}$

Figure 3: Example of a Rule Supporting the Assistance

expert advisor, is an implementation of various strategies which deal with the data distribution problem. The result of an application of these strategies would not be the final version of the data distribution, but a partial one which would serve as a good basis for gradual extension. Finding partial data distribution may be performed automatically. One such strategy will be described in Section 6.

Automatic Application of Optimizing Transformations. Investigating the program model, the expert advisor can detect program phases where better performance could be achieved. An improvement in performance can be often realized by an application of particular optimizing transformation(s) (loop interchange, loop distribution, etc.).

To imagine such situation, we suppose that parallelization of a program is in its second phase (after the P^3T run) and the expert advisor has identified the loop nest with the highest number of estimated cache misses. After analysis of the current situation, the expert advisor can decide to apply the loop interchange transformation to reduce the number of cache misses during realization of the loop nest.

As other examples demonstrating the automatic application of optimizing transformations, we consider a situation in the first step of program parallelization. The input program has been annotated by statements for data distribution and a loop carried true dependence has been identified in a loop. The block distribution has been proposed for an array identified as the carrier of dependence. The expert advisor detects, that the data dependence is a forward dependence. The loop distribution transformation is then applied by the expert advisor to remove the data dependence as well as the sequential program execution which it would cause.

5 What is Required To Realize An Expert Advisor

In this section we outline the structure of an expert advisor which is capable of the three variants of expertise as they have been described in previous sections. A simplified structure of the expert advisor can be seen in Figure 4. The five modules (interface module, expertise module, inference engine, parallelization model, knowledge base) can be identified. Brief description of each module follows in the next subsections.

5.1 Interface Module

The **interface module** of the expert advisor enables the exchange of information and control between XPA and the other tools integrated in VFCS parallelization environment. The module is composed of two submodules: **the expert advisor-user interface** and **the expert advisor-VFCS interface**.

In our prototype version we are not developing a natural language based consulting subsystem as the expert advisor-user interface. A simpler, but satisfactory, window based interface is used for expert advisor-user communication. For example, to realize the query phase of the assistance and guidance modules, the menu driven interface is proposed. All questions addressed to the expert advisor can be chosen from the menus in a flexible way. Particular answers and explanations are formulated in a nearly natural language. Various graphics displays are available to present actual information about the current parallelization state.

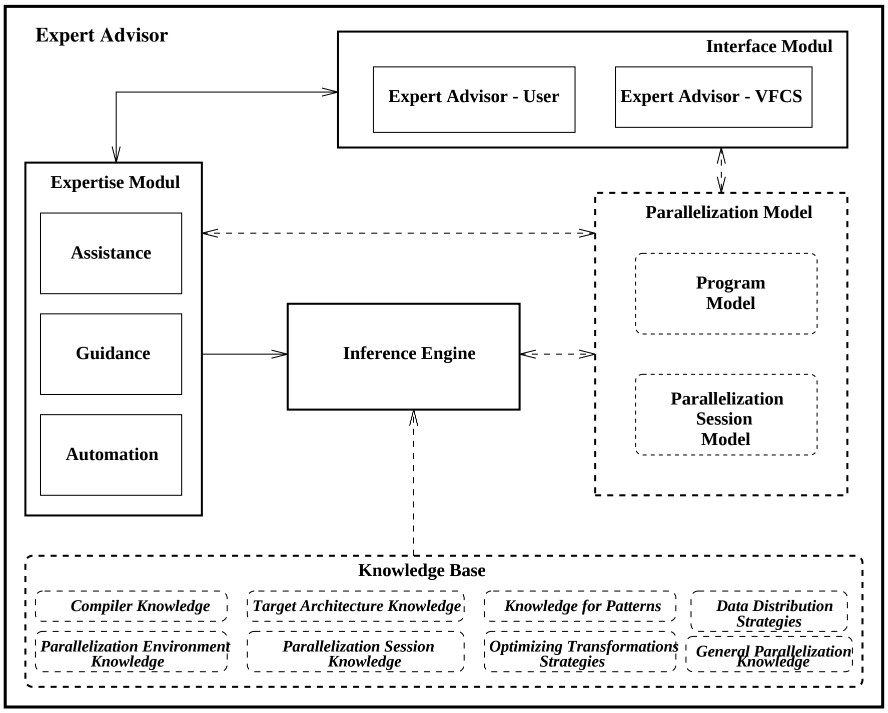

Figure 4: Structure of the Expert Advisor

All the tools which are integrated within the VFCS parallelization environment, and which we assume to be incorporated in our prototype *intelligent parallelization session* (the Weight Finder, the P^3T, the Pattern Matcher), are directly integrated with the compiler. Therefore, only the expert advisor-VFCS interface is required. This interface is fully implemented and will be described in 6.3.

5.2 Expertise Module

The expertise module of the expert advisor consists of three submodules each corresponding to one of the modes of expertise (assistance, guidance, automation), which were outlined in Sections 3 and 4. In this module all the procedural code, which realizes the control over the tool, is concentrated. The module is closely connected to the inference engine and to the interface module. It gathers actual information about the current parallelization state from the parallelization model.

5.3 Inference Engine

The inference engine is “the brain” of the expert advisor. Here, special knowledge picked out from the knowledge base, and particular facts valid in the parallelization model, are combined to deduce the consequences of known facts, or to find solutions that achieve the known goal. The **forward chaining** and **backward chaining** mechanisms (or a mixture of them) are used in our inference engine for application of rules from the knowledge base.

5.4 Parallelization Model

Before the human expert in the parallelization domain can utilize his deep parallelization knowledge, he must, first of all, acquaint himself with all important features of the input program and the current configuration of the parallelization environment. The expert creates "the program model" in his mind. As said before, a parallelization session is a complex iterative process. During this process the program is transformed from one form to an other. Three iterative cycles of program tuning have been distinguished. To be able to drive this complex process, the other model, the "parallelization session" model, has to be created in the expert mind. Moreover, the expert must keep track of the correlation between the two models in his mind.
Because we are modelling the human expert advising within parallelization, the both above mentioned models, **the program model** and **the parallelization session model**, are components of the expert advisor. In Figure 4, they can be seen as the building blocks of the **parallelization model**. In the expert advisor parallelization model, especially the features, which reflect the human view of the program and the parallelization session, are taken into account. Parallelization experience of the human expert plays the main role in selecting the abstraction level on which the program and the parallelization session features are formulated. Expert parallelization knowledge can be formulated only on the basis of concepts which form the program and session models.
Moreover, the program model must be dynamic to preserve its validity during parallelization session and to reflect the current state of the program and session after particular parallelization steps or optimization actions (e.g. loop transformations).

5.5 Knowledge Base

The parallelization model, as described above, is the source of concrete information about the program and the parallelization environment, and it reflects the progress in parallelization of the program. This model must be created and kept up-to-date every time a new program is parallelized. On the other hand, expert knowledge of the parallelization of various programs is static. The same knowledge can be used to parallelize different programs. Only if new parallelization knowledge has been acquired, does the knowledge base manager have to modify the knowledge base in order to integrate this new knowledge.

To support the professional parallelization expertise within the VFCS environment as discussed in Sections 3 and 4, the knowledge base of the expert advisor must include various qualitatively different aspects of parallelization. A necessary precondition is that the knowledge about different parts of this process is contained in a common base. A uniform representation is required to allow the utilization of fragmented, partially known solutions of individual parallelization steps, as have been developed in the past. Moreover, it must provide a framework for covering both human knowledge of parallelization (parallelization experience, heuristics, shortcuts) and the precise analysis and transformation techniques contained in current parallelizers. Hence, the implicitly imprecise, and often ad hoc, human solutions can be automatically improved by exploiting precise explicit methods, and vice versa.

We have distinguished the following top level *knowledge clusters* which the expert advisor knowledge base must consist of:

Parallelization environment knowledge. Instructions of how to use particular tool within VFCS parallelization environment, knowledge supporting decisions when a tool must be used, knowledge supporting the explanation of the reasons why the tool has to be used.

Compiler knowledge. This knowledge supports prediction of the effects caused by the compiler's parallelization transformations, e.g. where in the program communication statements will be inserted, what communication patterns will be generated, what the results of optimization of communications will be, where problems related to mask optimization will arise, etc.

Target architecture knowledge. This comprises abstraction of the target machine features into concepts that can be used by various heuristics, the communication characteristics of the various communication patterns (parameterized by the the data elements transferred through the communication network of the target architecture), the knowledge supporting the estimation of performance for representative classes of codes.

Parallelization session knowledge. This includes fixed scenarios of the parallelization and optimization steps, and knowledge supporting decisions which subscenarios are to be applied in the current parallelization state.

Knowledge for code patterns. For every code pattern, knowledge supporting decisions on parallelization proposals must be included in the knowledge base. Knowledge should be parameterized by array sizes, ranges of involved loops, and by the target architecture configuration. The parallelization proposals, such as efficient local data distributions or suitable optimizing transformations, as well as knowledge of how to estimate the performance of the pattern, help the expert advisor in providing expertise.

Data distribution strategies. This contains knowledge supporting selection of a data distribution type, decisions regarding the privatization of an array, or replication of an array dimension. Other knowledge supports the interpretation of performance parameters obtained from performance prediction and analysis tools, and captures the consequences of these parameters on the data distribution. Various supporting data distribution strategies, which can be applied automatically, are included in the knowledge base, as well as knowledge supporting decisions regarding redistribution of data.

Strategies for application of optimizing transformations. Contains: knowledge supporting the identification of program phases, which will give poor performance on the target architecture; knowledge supporting the deduction of reasons for poor performance of the program phase on the target architecture; knowledge supporting selection of appropriate optimizing transformation(s) to refine the performance of the program phase on the target architecture.

6 Current State of the Expert Advisor Development

In this section we describe the implementation of the Expert Advisor prototype and show its functionality on an application example: the simplified automatic generation of program data distributions. The implementation was realized with the following two goals: first, demonstration of the suitability of the expert system approach for supporting parallelization within VFCS, and second, the creation of a flexible and expandable implementation skeleton for future extensions. This skeleton comprises the essential components, and it is being gradually extended and refined. At this point, we would like to note that the implementation effort described in this section has not been targeted only to the particular problem presented; it aims not to restrict future work. As mentioned, the final goal is to support the entire VFCS parallelization session and to make the parallelization process as automatic as possible.

6.1 XPA Implementation Requirements

The **Expert Advisor** (XPA) was designed as a *separate subsystem* of the VFCS parallelization environment with a precisely specified interface to other components of the environment [20]. Among other things, this enables realization of XPA using a specialized knowledge representation and inference tool - an *expert system shell.* The selection of an appropriate shell for the XPA implementation was influenced by the following major requirements resulting from goals that XPA aims to achieve (we discuss them below):

- flexibility of the interface to other tools and to users
- implementation of large and complex knowledge base
- advanced control and inference mechanisms

The XPA interface to other components of the parallelization environment, to the end-user and to the knowledge base developer must be *flexible* enough to allow *gradual evolution* of XPA from a system with intensive user interaction to a largely independent automatic system. The interactive mode of interface is important especially for the development and refinement of a knowledge base in situations where several or most of the relevant aspects of parallelization are not yet understood or are only vaguely specified. This permits easier experimentation with the system and acquisition of the relevant knowledge. As soon as the experimentation yields a fixed and refined procedure for solving a particular problem, this stable procedure is incorporated into a knowledge base in the form of rules, and the interface to the user is replaced by the corresponding interface routines, which gather the necessary information from other components of the parallelization environment without user assistance. Based on this experimentation, the interface routines are also gradually enhanced.

The XPA consists of two main components: a *knowledge base* and an *inference engine.* The knowledge base comprises *objects* and *rules.* The objects represent parallelization-related facts and concepts of various abstraction level. The inference engine performs reasoning over the objects using the rules to derive new facts and/or to find a solution of a particular parallelization problem.

The XPA knowledge base contains very different components and it is expected to grow considerably because of the complexity of the parallelization problem domain. A flat organization of the knowledge base with simple objects and rules does not suffice in this situation. Therefore, the

XPA's knowledge base has to be hierarchically organized, and it makes use of complex, structured objects and rules. Hence, it is obvious, that an *advanced expert system development environment* (shell) had to be selected as the XPA implementation substrate, in order not to become a restrictive factor in the future XPA evolution.

6.2 Implementation Substrate Features

The **ProKappa Expert System Development Environment** from IntelliCorp [28] has been chosen for the XPA implementation. This professional environment meets our design requirements. We describe some of its most important features and outline their exploitation in the framework of XPA:

- both ProKappa and VFCS are **C programmed systems**. This greatly facilitates their integration. In particular, direct access to internal VFCS data structures and procedures is easily achieved, and C procedures may be used to realize parts of XPA when appropriate.
- The ProKappa environment incorporates a full and effective implementation of the **object-oriented paradigm**. ProKappa's object system supports a natural, flexible, modular, and expandable representation of XPA information components, such as program constructs (do-loops, statements), data and control flow information, data dependencies, transformations, performance parameters, data distributions, and target architecture features. The *methods* and *monitors* are used in automatic filling and checking of values in object slots.
- **Active Relations** allow specification of simple local relations among objects on the slots level. They are convenient in the framework of XPA to accomplish automatic local information distribution and classification, mainly for preprocessing of information obtained from other subsystems, e.g. for classification of array access patterns for a selected do-loop construct from the code currently being processed.
- The shell's **inference engine** supports both *forward* and *backward chaining* of rules, with *automatic backtracking* and searching for alternative solutions. The **ProTalk** descriptive language allows the specification of rules in a humanly readable form. Pattern matching drives the application of rules. Rules are grouped into **rulesets**; the invocation or disabling of each ruleset can be explicitly controlled. In XPA, the ruleset mechanism is used to split the large solution space into several smaller ones, thereby reducing the solution complexity and enhancing modularity. The focus of the solution is moved according to the current subgoal by enabling or disabling particular rulesets.
- **Developer** and **End User Interface** tools support not only a comfortable environment for the creation and testing of ProKappa applications, but also facilitate construction of the interactive interface between XPA and the user of the parallelization system. The graphical developer interface comprises a convenient set of **workbenches**, **browsers**, and **probes**: C Workbench, ProTalk Workbench, Interface Workbench, Application Browser, Object Browser, Slot Properties Probe, Value Type Probe, etc.

6.3 Integration of XPA into VFCS Parallelization Environment

As was mentioned, the VFCS Parallelization Environment is centered around the parallelizing compiler and it comprises a set of performance analysis subsystems and external tools. An integration of XPA with any particular subsystem of the parallelization environment constitutes two levels:

- physical integration (interconnection)
- logical integration

The **physical integration** of XPA and other component of parallelization environment has to establish a low level data and control flow among cooperating subsystems. During implementation of the XPA prototype, we concentrated on the interconnection of XPA and the VFCS compiler since the invocation of most parallelization support tools results in an annotation of the internal VFCS structures representing the processed program. Hence, the VFCS can serve as a mediator between XPA and a support tool. As a rule, the data and control flow are bidirectional. Using ProKappa for XPA implementation greatly facilitates the XPA-VFCS integration since ProKappa allows direct loading or linking of modules programmed in the C language. The XPA-VFCS interconnection includes also an integration of XPA's graphical user interface with that of VFCS. The VFCS memory management had to be adapted according to the ProKappa requirements to allow XPA-VFCS interconnection.

By the **logical integration** of XPA and other tools of the parallelization environment we mean the mechanisms for extraction and exploitation of relevant information obtained from the tool. Such information has to be explicitly represented inside XPA or it has to be dynamically obtained from the tool if it is to be used within XPA's rules. Therefore, the low level of logical integration includes routines that access the required information inside the tool and convert it into suitable ProKappa representation.

Currently, the XPA prototype is integrated with the VFCS compiler and with the Weight Finder as a representative of performance analysis tools. The integration with the Pattern Matcher (cf. Section 2.2.3) is in progress.

The XPA prototype is available in two modifications. The first one, the **Development Version** of the XPA system, is dedicated to the interactive development and debugging of the XPA. Here, the VFCS object modules are loaded into the ProKappa Development Environment as can be seen in Figure 5. The shadowed boxes correspond to the ProKappa development and debugging facilities. The second, the **Run-Time Version** of XPA (Figure 6), does not contain development and debugging facilities, and is targeted to the end user of the VFCS system. It can be delivered as an integral part of VFCS system. The VFCS modules are linked together with the ProKappa run-time modules and compiled XPA modules.

6.4 Program Model

The XPA proposal requires the definition and representation of parallelization related concepts at very different abstraction levels (cf. Section 5.5). Some of them are closely related to the processed program and performance parameters, while others are needed for incorporation of the heuristics and for driving a parallelization session. Note that the existence of explicitly represented concepts is a necessary condition for specification of XPA rules.

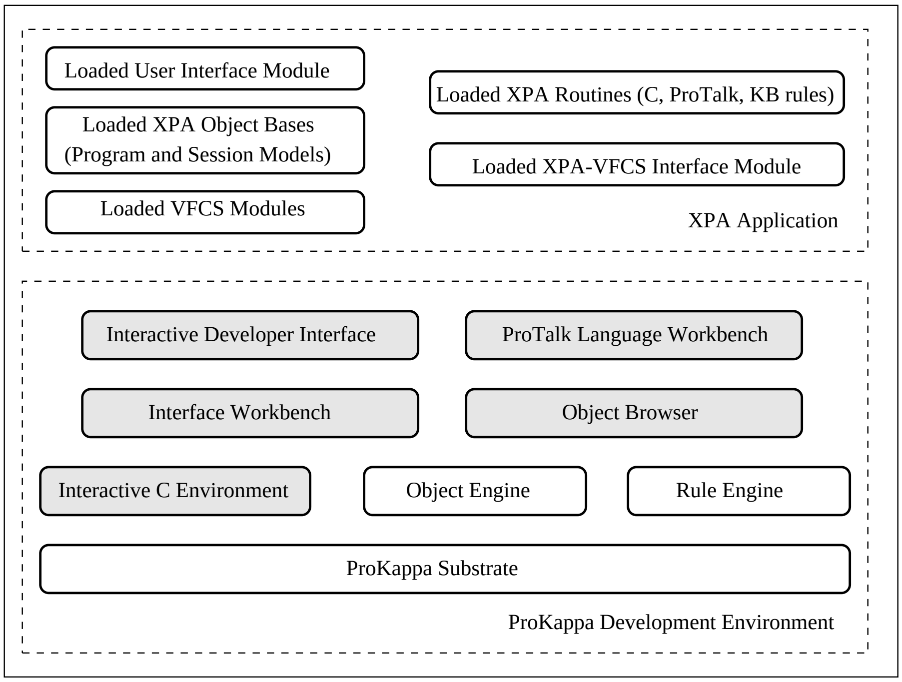

Figure 5: XPA Development System Components

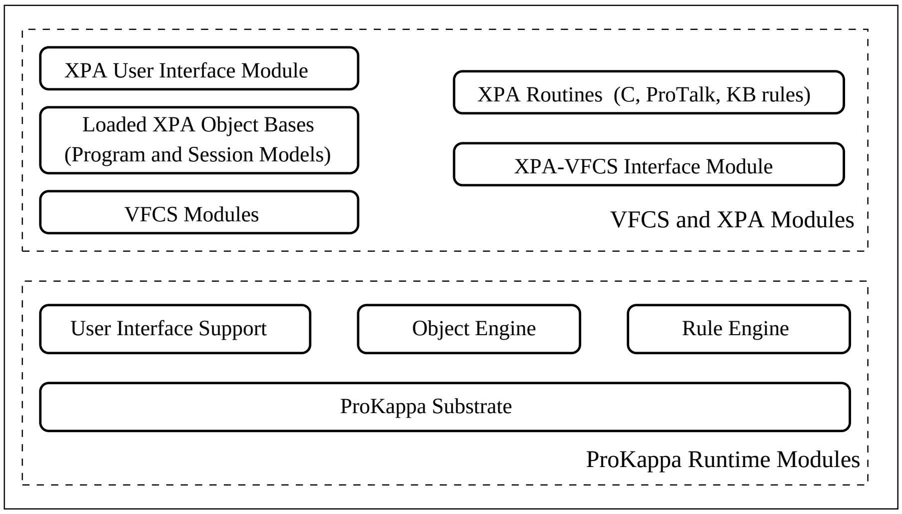

Figure 6: XPA Run-Time System Components

Within ProKappa, the concepts are represented by structured *objects*. Objects can either represent templates for sets of real world entities or individual entities. So, the ProKappa system has two kinds of objects, *classes* and *instances*. Classes and instances are organized hierarchically so that information specified in a class is inherited by its instances. Both classes and instances have *slots* which represent characteristics or attributes of objects. A slot contains either data (*values*) or a pointer to a function (*method*) which performs some action associated with the object. A slot containing data may have associated routines (*monitors*) that, for instance, validate the slot's value (*value checks*), implement *inverse relations*, and realize various calculations accessing values from other slots (*slot formulae*). A slot's value may refer to other objects, so e.g. complex graph-like structures of objects may be easily represented.

In the XPA prototype implementation, we prefered to concentrate on the representation of concepts related more or less directly to a program. Such concepts are at the relatively low level of abstraction. We use the term *program model* to denote the appropriate part of the XPA object base. We claim that the program model is an essential part of the XPA knowledge base because:

- The XPA program model representation can be directly connected to the VFCS internal program representation. XPA needs access to VFCS's program representation to acquire information about the program and to perform proposed parallelization actions without intervention of the user. For instance, optimizing transformations usually requires the specification of the program portion on which they are to be applied.
- The majority of the low and medium level expert rules refer to the program, e.g. they test program structure, look for data dependences and particular array accesses.
- The program model can reflect different stages of the program parallelization and it forms a natural framework for placement of recognized code patterns and performance indices.
- The selection of the most appropriate parallelization strategy is significantly influenced by the program complexity, its structure, and decomposition into phases.

Currently, about two dozen ProKappa object classes are used to form the XPA program model. The particular program portions are represented within XPA program model as objects that are instances of appropriate object classes.

The internal representation of the original sequential program in VFCS, after the VFCS frontend processing, and after execution of the **Weight Finder** profiler [13], is the basic information source for the XPA. This internal VFCS representation offers XPA a normalized form of the original program. The relevant segments of such VFCS internal representation are converted into ProKappa objects by Initial Program Representation module to form an important part of the XPA program model.

The realization of this XPA module comprised the proposal and implementation of ProKappa objects for basic XPA representation of the input serial VFCS program, together with the methods and monitors that create particular objects, extract the relevant information for parallelization from the internal VFCS structures, convert it into suitable form, fill particular slots of XPA objects, and build the representation structure by establishing interconnections among XPA objects.

This part of the XPA program model is obtained by preprocessing, compressing, and reorganization of the information obtained from the symbol table, syntax tree, flow graph, and from precise data flow and data dependence analysis. The objects of this group can be considered as static and they are normally created in a separate phase at the beginning of XPA's processing.

We present here the main object classes that corresponds to the *initial program representation* part of the program model. They are divided into three groups according to what they represent in the model[3]:

- **ARRAYS**
  The following object classes are used to represent arrays declared and accessed within program or subroutines. Dimensions of a declared or accessed array are explicitly represented. Loop carried array dependences are grouped according the loops that are carriers of dependences.
  - *array;* Slots: type (adjustable, constant, parameter, common), identifier, rank, dimension list.
  - *array dimension;* Slots: array, dimension number, bounds, array dimension occurrences.
  - *array occurrence;* Slots: array, source and sink of dependences, usage(def, use, both), cost-relevant statement, array dimension occurrences, pointer to corresponding VFCS information.
  - *array dimension occurrence;* Slots: array occurrence, array dimension, subscript type (constant, simple, constant linear, linear, unrecognized), subscript, align preferences.
  - *dependence*; Slots: type (true, anti, output), do-loop relevant statement (dependence carrier), sink and source of dependence (array occurrences).
- **SUBROUTINES**
  This group of object classes represents a program, its units and call graph.
  - *program;* Slots: file, units, state (loaded in VFCS, Weight Finder information available, represented in XPA, annotated, parallelized in VFCS, ...).
  - *program unit;* Slots: name, called in, called units, formal arguments list, declared arrays, parallelizable components, relevance (computed from Weight Finder information), strategic level, data distribution hypothesis, status.
- **UNIT STRUCTURE**
  The syntactic and control structure of a unit is represented by interconnected objects representing the unit's relevant statements. Cost-relevant statements contain an array reference. The program is also decomposed into parallelizable components (phases).
  - *relevant statement;* Two types of relevant statements are recognized by XPA:
    - *CALL relevant statement;* Slots: actual argument list, previous statement, next statement, outer statement, relevance, pointer to corresponding VFCS structure.
    - *DO-LOOP relevant statement;* Slots: loop variable, loop bounds characteristic, loop bounds, align preferences (SAC's), inner statements, previous statement, next statement, outer statement, relevance, pointer to corresponding VFCS structure[4].

[3] For the sake of simplicity, we will present informally only the most important value slots for every class, and we will not include methods and monitors.

[4] For illustration, the Figure 7 shows the excerpt from XPA's object base for the DoLoop object class.

Class DoLoop

Parent → RelevantStatement
Slot **LoopVariable** → ?
Slot **Type** → Normal
Monitor Facet **WhenChangedFacet** → ValueTypeMonitor@ValueType
SelfFirstUnion Facet **ValueType** → [(OneOf, Normal, Forall), Alarm, DialogBox]
MVOverride Slot **CarriedDependences** → ?
Facet **SlotInverse** → DoLoop
Monitor Facet **WhenChangedFacet** → ValueTypeMonitor@ValueType, SlotInverseMon@SlotInverse
SelfFirstUnion Facet **ValueType** → [(Instance, Dependence@), Allow, Exception]
Slot **UpperBoundCharacteristics** → Unknown
SelfFirstUnion Facet **ValueType** → [(OneOf, Constant, Variable, Expression, Unknown), Alarm, DialogBox]
Monitor Facet **WhenChangedFacet** → ValueTypeMonitor@ValueType
Slot **SAC** → ?
SelfFirstUnion Facet **ValueType** → [(Instance, SAC@), Alarm, DialogBox]
Monitor Facet **WhenChangedFacet** → ValueTypeMonitor@ValueType, SlotInverseMon@SlotInverse
Facet **SlotInverse** → RelevantStatement
Slot **LoopBoundsPrecision** → Unknown
SelfFirstUnion Facet **ValueType** → [(OneOf, Unknown, Exact, Estimated, Profiled), Alarm, DialogBox]
Monitor Facet **WhenChangedFacet** → ValueTypeMonitor@ValueType
MVOverride Slot **CostRelStmts** → ?
Slot **UpperBound** → ?
Slot **FirstInnerSegment** → ?
MVOverride Slot **InnerSegments** → ?
SelfFirstUnion Facet **ValueType** → .
Monitor Facet **WhenChangedFacet** → .
Method Slot ‘"**FillBoundsInfo!**" → !DoLoop_FillBoundsInfo
Slot **LowerBoundCharacteristics** → Unknown
SelfFirstUnion Facet **ValueType** → [(OneOf, Constant, Variable, Expression, Unknown), Alarm, DialogBox]
Monitor Facet **WhenChangedFacet** → ValueTypeMonitor@ValueType
Slot **LowerBound** → ?
Slot **Classification** → ?
Slot **DependenceType** → Unknown
SelfFirstUnion Facet **ValueType** → [(OneOf, None, Unknown, Serializing, Uncomputable, LoopCarried), Alarm, DialogBox]
Monitor Facet **WhenChangedFacet** → ValueTypeMonitor@ValueType
Slot **LocSerialNum** → ?
Slot **RelatedCRelStmt** → ?
SelfFirstUnion Facet **ValueType** → [(Instance, CostRelStmt@), Alarm, DialogBox]
Monitor Facet **WhenChangedFacet** → ValueTypeMonitor@ValueType, SlotInverseMon@SlotInverse
Facet **SlotInverse** → RelatedRelStmt
Slot **LexNextSegment** → ?
SelfFirstUnion Facet **ValueType** → .
Monitor Facet **WhenChangedFacet** → ValueTypeMonitor@ValueType
Slot **LexPreviousSegment** → ?
SelfFirstUnion Facet **ValueType** → .
Monitor Facet **WhenChangedFacet** → ValueTypeMonitor@ValueType
Slot **OuterSegment** → ?
SelfFirstUnion Facet **ValueType** → .
Monitor Facet **WhenChangedFacet** → .
Method Slot ‘"**CreateInstance!**" → !DoLoop_CreateInstance

Figure 7: XPA's DoLoop Object Class

- *cost-relevant statement;* Slots: type (if, assignment, loop, call, unrecognized), related relevant statement, array occurrences.
- *parallelizable component;* Slots: type (atomic, call, composed, strategic), relevant statement, included in parallelizable component, includes components, relevance, inlining recommendation flag.

The program model objects mentioned above are directly related to the original program undergoing parallelization. The XPA then preprocesses the initial program representation objects to generate the program model objects for the XPA solution phase. We use the term *intelligent preprocessing* to denote this enhancement phase of the program model. The objects created during intelligent preprocessing by an extensive application of rules from XPA's knowledge base belong to the following object classes groups:

- **PARALLELIZATION STRATEGY**
  The solution space of the parallelization problem is huge, so an exhaustive examination of it is unrealistic, especially for larger programs. For instance, the determination of an optimal data distribution requires making trade-offs for distribution proposals as derived locally for separate portions of the program code to achieve globally optimal data distribution. The automatic data distribution problem is known to be NP-complete. Thus in practice, we must look for clues in the code which can help us to narrow the search as far as possible. The important part of parallelization strategy is related to the program structure: which portions of the code will be visited and in which order. Several solution strategies may be applied. The optimal parallelization strategy depends on the processed program (its size, structure, computational intensive parts) and a parallelization problem to be solved. In XPA, the object *parallelizable level* was defined to support the realization of various strategies. The parallelization components are grouped into parallelizable levels by a parallelization strategy generation algorithm. This grouping is also done interprocedurally; the hierarchical structure of parallelizable levels together with the introduced ordering of sublevels define a parallelization strategy. For illustration, Figure 8 presents the XPA decomposition of the Jacobi program into parallelizable levels. A level may also be used to encapsulate selected program parts and to force their separate and/or preferential processing. The second purpose of introducing of parallelizable levels, that we do not discuss further here, is enabling of (automatic) determination and placement of redistribution points, i.e. the program points where some program arrays may be potentially redistributed.

  - *parallelization level;* Slots: type (collect, create, update, extend), building components, parent level, sublevels, hypothesis, propagation flag, state.

- **ALIGNMENTS**
  The parallelizable levels are further preprocessed; based on the form of subscripts in array occurrences within corresponding parallelizable components, align preferences are recognized and represented. Alignment and distribution conflicts are then recognized and classified. XPA use the object class *set of array dimension candidates for partitioning (SAC)* to represent local align preferences.

  - *align preferences (SAC);* Slots: relevant statement, array dimension occurrences.

- *inter-dimensional alignment conflict;* Slots: array alignments in conflict. This type of alignment conflict refers to dimensions of different arrays.
- *intra-dimensional alignment conflict;* Slots: type (shift, unrecognized), array dimension occurrences. This type of conflict represents different requirements for alignment within a dimension of an array.

Finally, a data distribution proposal is generated during the *data distribution extraction* phase. In general, within the framework of the program model as described above, we see the data distribution extraction as a process of gradual creation of a *data distribution hypothesis.* The hypothesis generation algorithm may be realized as a recursive process with incorporated backtracking. In order to resolve distribution conflicts, the algorithm may apply various *heuristics.* The algorithm proceeds according to the parallelization strategy defined in the previous phase, and the distribution hypothesis is constructed by an application of *local parallelization methods* realizing selected *parallelization tactics.* The data distribution hypothesis defines the alignment and distribution of array dimensions to a virtual processor array; an adaptation of the hypothesis for actual architecture (mapping) gives the resultant data distribution.

However, the currently implemented distribution algorithm is simpler. It does not use heuristics to resolve conflicts. Instead, it does not distribute array dimensions that are involved in most kinds of alignment conflicts, and it avoids distributions causing a loop serialization. Nevertheless, the following object classes have been prepared for a more general version of the algorithm:

- **DISTRIBUTION HYPOTHESES**
  The distribution hypothesis object is coupled with a particular parallelizable level and it represents the current (temporal, working) state of a distribution proposal for that level. The substantial part of a distribution hypothesis is a set of dimension distribution hypotheses that contain distribution proposals for separate dimensions of a virtual processors array.

  - *distribution hypothesis;* Slots: parallelizable level, cost, dimension distribution hypotheses, state.
  - *dimension distribution hypothesis;* Slots: distribution hypothesis, involved array dimension occurrences, array dimensions, original align preferences, unresolved conflicts.

- **DATA DISTRIBUTIONS**
  The Vienna Fortran language uses the DIST and PROCESSORS statements for the specification of array distributions. The corresponding XPA object classes (see below) are designed for:

  1. The representation of the specified data distribution if the input program is (partially) annotated to allow reasoning about such distributions;
  2. The representation of array distributions extracted from a final distribution hypothesis to allow annotation of the program with appropriate distribution statements.

  - *array distribution;* Slots: array name, list of dimension distributions, extracted from distribution hypothesis, processor shape.
  - *array dimension distribution;* Slots: type (block, cyclic), block characteristics, dimension num, number of processors in dimension.
  - *processors shape;* Slots: identifier, dimensions list.

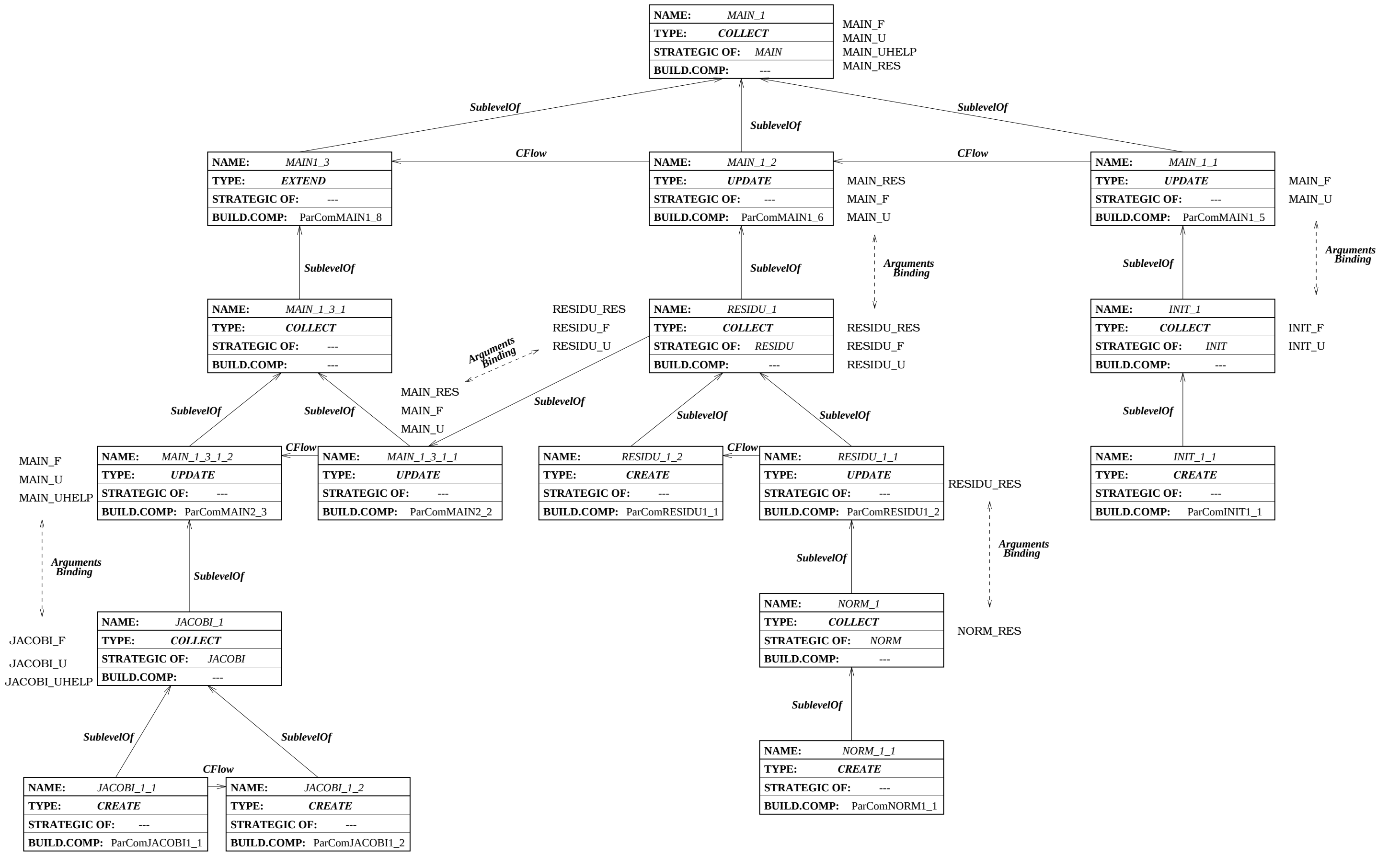

Figure 8: The Level Graph for Jacobi Program

6.5 Knowledge Base - Rules

The knowledge base rules form the most substantial part of XPA. Unlike the program model objects that represent concrete facts concerning the particular program processed and the status of the solution process, the knowledge base rules may represent more general information that can by applied during parallelization of various programs.

The knowledge base rules are not static; they are updated and refined by a knowledge administrator, when new knowledge is formulated, or when experimentation with the tool has shown that modification of the knowledge is required. By the application of the rules, various aspects of the parallelization process can be supported. The rules are clustered according to program parallelization subproblems.

In ProKappa, the knowledge clusters correspond to rulesets[5]. It is possible to control application of rules by specifying appropriate rulesets which should be used, a way of reasoning over rules (forward, backward chaining), and significance of rules (their priority). During the solution, the knowledge base rules are used to deduce new facts from old ones, to recognize situations for which an appropriate solutions exist, or to provide the system with heuristics for solving problems arising during a parallelization process.

The rules currently present in the XPA knowledge base are used mainly when building the XPA program model. Hence, they are at a relatively low level of abstraction. Most of the current XPA rules are applied during the intelligent preprocessing and data distribution extraction phases. For instance, rules are used for:

- spreading consequences of true and anti dependences found in the program
- finding align preferences for dimensions of program arrays
- identifying inter and intra-dimensional conflicts in proposed dimension alignments
- solving the alignments conflicts found

In XPA, the corresponding rulesets[6] are utilized:

- DependenceConsequences
- AlignPreferences
- AlignmentConflicts
- HypothesisConflicts

The rules in these rulesets are forward chaining rules. The forward chaining mechanism is usually convenient to derive new facts from the old ones by inductive reasoning, as opposed to the backward chaining that is normally applied to decompose a problem into subproblems using deductive reasoning. During forward chaining, ProKappa attempts to find consequences of known facts. During backward chaining, the system attempts to find values that cause particular consequences. The forward chaining over the above mentioned rulesets derives the following new facts:

[5] The ProTalk descriptive language is used to write rules.

[6] Using ProKappa's terminology, a ruleset corresponds to a named set of rules that can be processed separately.

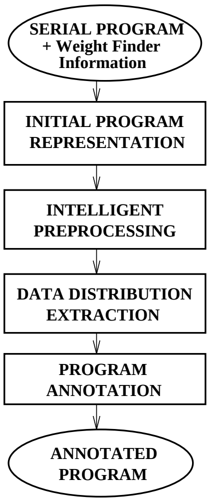

Figure 9: XPA's Main Processing Steps

- dimensions of arrays which, when distributed, would cause the serializing of program execution
- array dimensions whose distribution would cause the communication among processors
- proposals for alignment of array dimensions on the same processor dimension
- conflicts in alignments of array dimensions; the solutions of these conflicts are proposed

On the basis of the new facts deduced by knowledge base application, the parallelization hypothesis, comprising the proposal for partitioning of program arrays, is constructed. The determination of the number of processor dimensions then follows. The file, containing the proposed program data distributions (expressed in the Vienna Fortran data distribution constructs), is then created. The next parallelization steps can then take place.

6.6 XPA Session - User's View

In this section we briefly describe a typical session for the Development Version of the XPA from the end-user and developer point of view. The description will be illustrated by several snapshots taken from the XPA session at different phases of its work. For an overview of the XPA development environment, see Figure 10. More about the contents of the figure will be given later.

Figure 9 summarizes the main phases of the XPA processing. The XPA input is the sequential Fortran program *without* the distribution specifications of program arrays. The input program may be completed with profiling information obtained by Weight Finder. The user invokes the VFCS and XPA actions over the input program by the selection of appropriate XPA menu items[7]. The XPA output is the current version of the input program in the form of an internal VFCS representation annotated by the proposed data distributions. Then the user, supported by VFCS, performs the

[7] The most of such separate actions can be integrated into the single procedure invoked by simple menu selection. However, in the current XPA version, they have been left separate to allow easier debugging and development of XPA.

final steps to create the target parallel program: automatic generation of communication statements for proposed data distribution, communication and masking optimization, and generation of the parallel program for selected target architecture [8]. A more detailed description of the whole XPA parallelization procedure is presented in the following paragraphs.

First, the ProKappa expert system development environment has to be running. The XPA application is then loaded into ProKappa. Now, the VFCS and XPA code together with data structures are inside the ProKappa working memory. After the selection of ProKappa `Run` button, the VFCS window appears on the screen. The main XPA menu is accessible through the `For Tests` item of the VFCS menu.

The user then loads the input serial program that is to be parallelized by specification of its name and directory within the main XPA menu and by pressing `Load Program` button. Inclusion of Weight Finder information can also be selected here (click the `VF Run` check button).

This is usually followed by an initial improvement of the input program structure to allow more precise successive data dependence analysis. At this stage of the XPA implementation, the user performs this step manually by invocation of suitable VFCS transformations such as intra and inter-procedural constant propagation, induction variable substitution, expression simplification, dead code elimination, and scalar forward substitution. Making use of VFCS `Transformations` menu, the user can apply these transformations selectively on particular parts of the code, or he can apply transformations globally on the whole code by using the XPA `Initial Program Improvement` submenu[9].

The next step is the creation of the initial input program representation as described in Section 6.4. Clicking the `Create XPA Initial Representation` button creates the initial representation for the whole program. Figure 11 illustrates the state after the creation of the initial program representation. The Object Browser shows the part of the initial representation object base.

At this point, the end-user should skip actions described in this paragraph and continue directly with the parallelization steps described in the next paragraph. The situation is different for the XPA knowledge base developer. Typically, the development cycles of the knowledge base are performed mainly at this point. The development includes testing, completion, and refinement of the knowledge base. The knowledge base developer can interactively use an exhaustive set of built-in ProKappa development tools. Moreover, the additional set of special XPA development tools has been created. For example, it includes the tools for saving and restoring the current state of the XPA object base, and the tools for visualization of appropriate segments of source code, that correspond to the particular XPA objects, in the VFCS main source text window. Note that the knowledge base development is a complex process and it is not intended to be performed by the naive XPA user. Figure 10 presents the snapshot of a typical XPA development session. Besides the VFCS window and the main XPA menu, the specialized ProKappa workbenches and the object browser are on the screen. The left hand side icons hide the additional ProKappa browsers and probes.

The `Program Parallelization` button displays XPA's submenu for driving the next parallelization steps. First, the program is decomposed into parallelizable levels (cf. Section 6.4) to define a parallelization strategy. We have now implemented the *Incomplete Relevance Guided Strategy* [1]. The main goal of this strategy is to concentrate only on the most important program code sections during a search for good data distributions, and therefore achieve a substantial reduction

[8] These final steps might be performed also automatically from XPA.

[9] In the future, the manual initial program improvement is intended to be replaced by the automatic one.

in the amount of conflicts which should be taken into consideration during the data distribution extraction. The parameters obtained from the Weight Finder are the most important information supporting an automatic construction of this strategy. As an option, the parallelization strategy can also be specified manually whereby the XPA drives the user through the specification and asks the user to provide missing data or information about the program. Figure 12 illustrates the status of XPA after the specification of the parallelization strategy. Note that new objects corresponding to parallelization levels have been generated (see the levelization protocol in the bottom window).

The snapshot in Figure 13 shows the XPA state after a data distribution was extracted. The bottom window shows (partly) the performed steps: the generation of alignment preferences and alignment conflicts, propagation of conflicts, and distribution extraction.

The final step is an annotation of the program by the data distribution selected. The VFCS window in Figure 14 (partly) shows the resulting annotated program using the Vienna Fortran syntax (note PROCESSORS and DIST statements). The annotated program now can be compiled within VFCS for the selected target machine. The `Parallelize in VFCS` menu item provides the user with pre-defined steps for generating the target parallel program.

7 Future Work

In this section we introduce our vision of an advanced programming environment which exploits the knowledge-based approach. We then give some extensions to our system which could be achieved in the short term. We conclude with a brief summary of the features.

7.1 Research in Knowledge-Based Parallel Program Development

From the user's point of view, **fully automatic compilation** of high-level programs (*without* additional directives such as provided by HPF) is a desirable goal. However, even in advanced current compilation systems such as VFCS, only a moderate amount of automatic support is provided; most strategic decisions must be made explicitly by the user. The major reason lies in the complexity of modern high-performance computers (HPCs) and the resulting difficulty of optimally tuning a given application program to the architecture.

Automatic compilation implies *automatic data distribution* and *automatic selection of transformation strategies*. While a data distribution strategy primarily specifies the critical work distribution and communication overhead, program transformations enable the detection, extraction, and exploitation of parallelism. Program restructuring may induce a variety of performance tradeoffs which need to be carefully evaluated. This requires in-depth knowledge of the application domain, programming paradigms, transformation system, data distribution schemes, and target architecture specific details. These parameters create a huge search space for parallelization and optimization strategies; as a consequence, even relatively simple subproblems of automatic compilation are either undecidable or NP-complete.

Taking the present state-of-the-art and the foreseeable development of compilation techniques into account, *fully* automatic compilation for HPCs does not seem completely feasible, even in the long term. However, successful research in this area at Vienna and elsewhere has produced a set of partial results which can be used as the basis for a pragmatic solution that will asymptotically approach the desired objective. Recent work includes an expert-system based prototype development of an automatic data distribution tool at Vienna (in the framework of XPA, as outlined

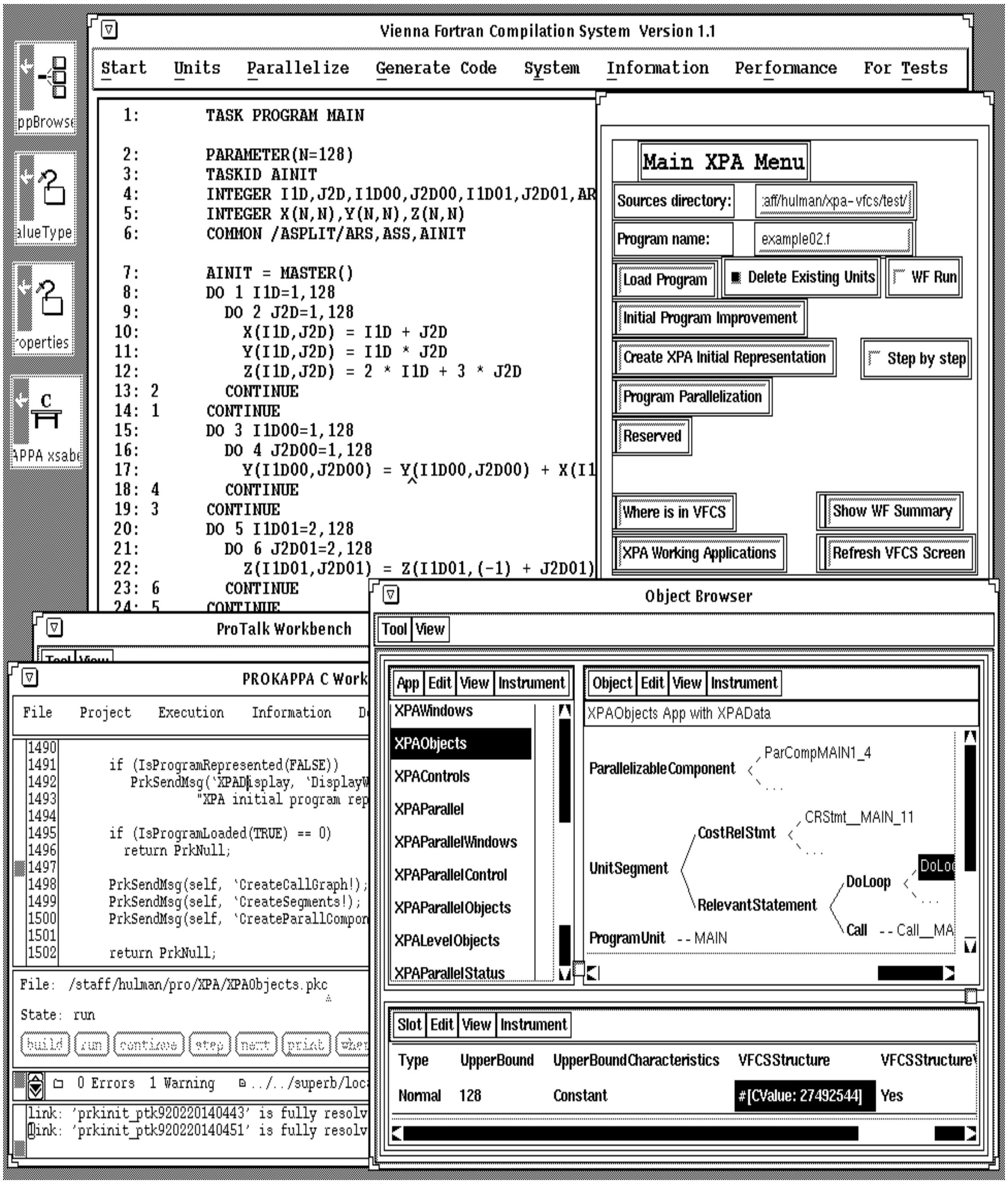

Figure 10: XPA's Development Environment

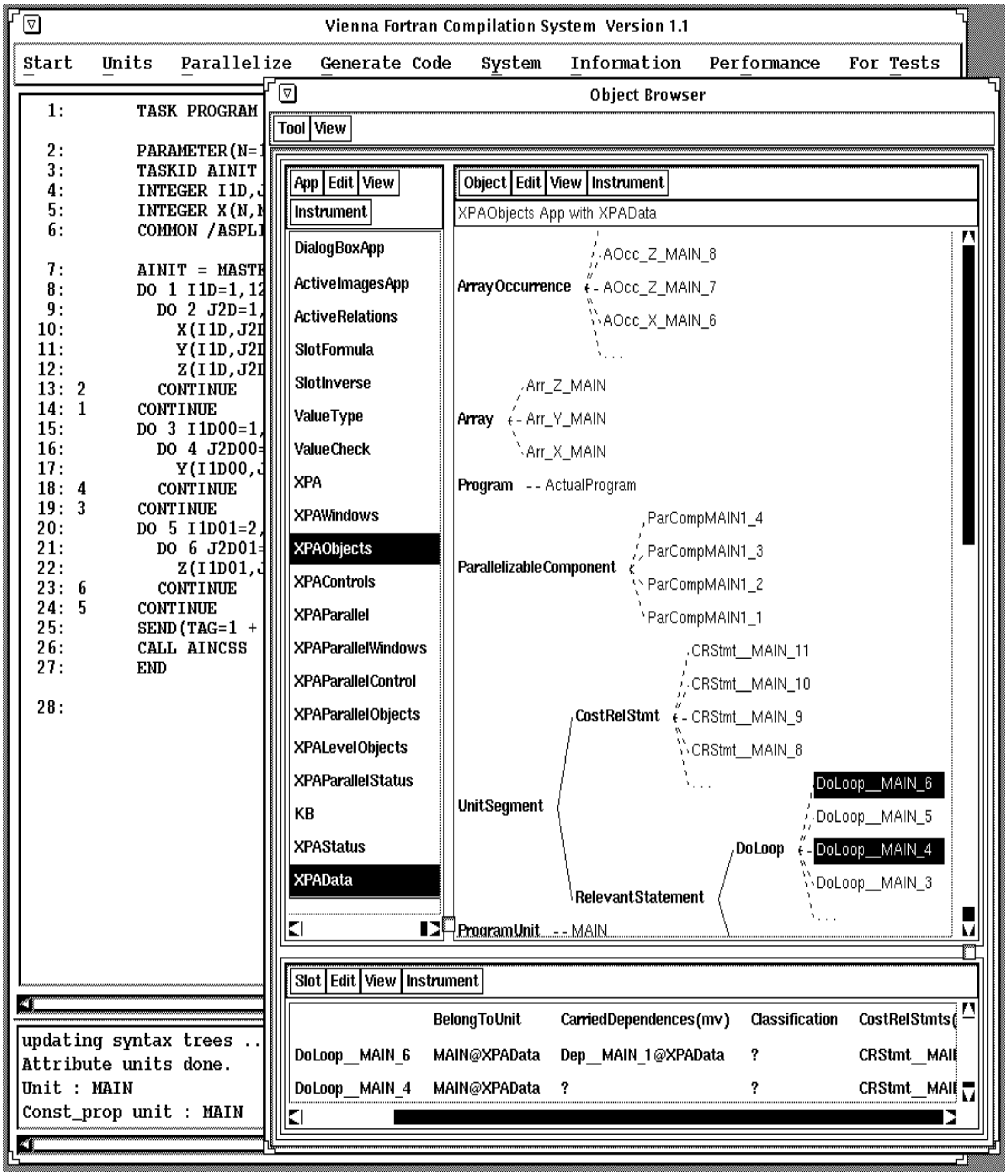

Figure 11: Initial Program Representation

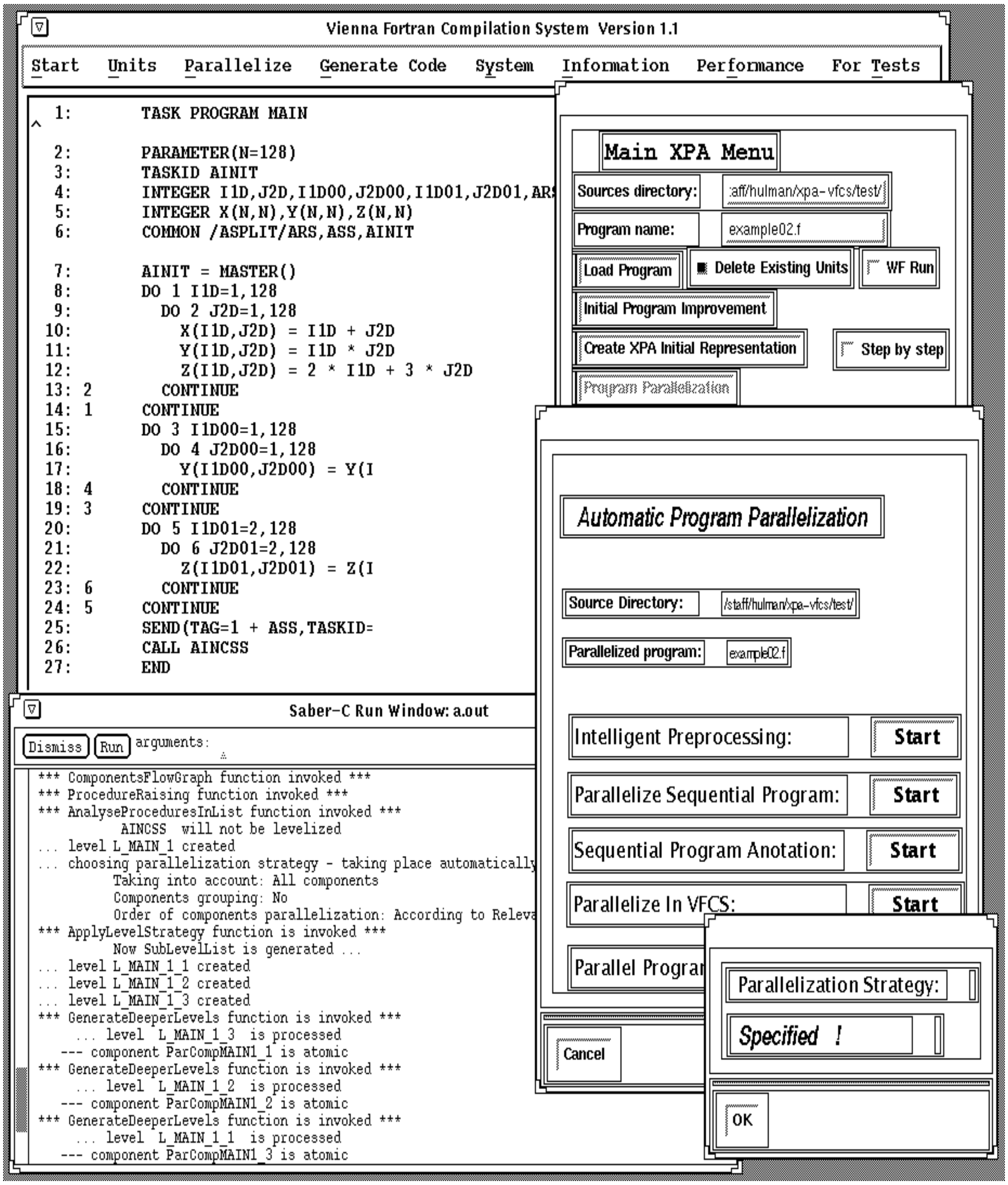

Figure 12: XPA After Generation of Parallelization Strategy

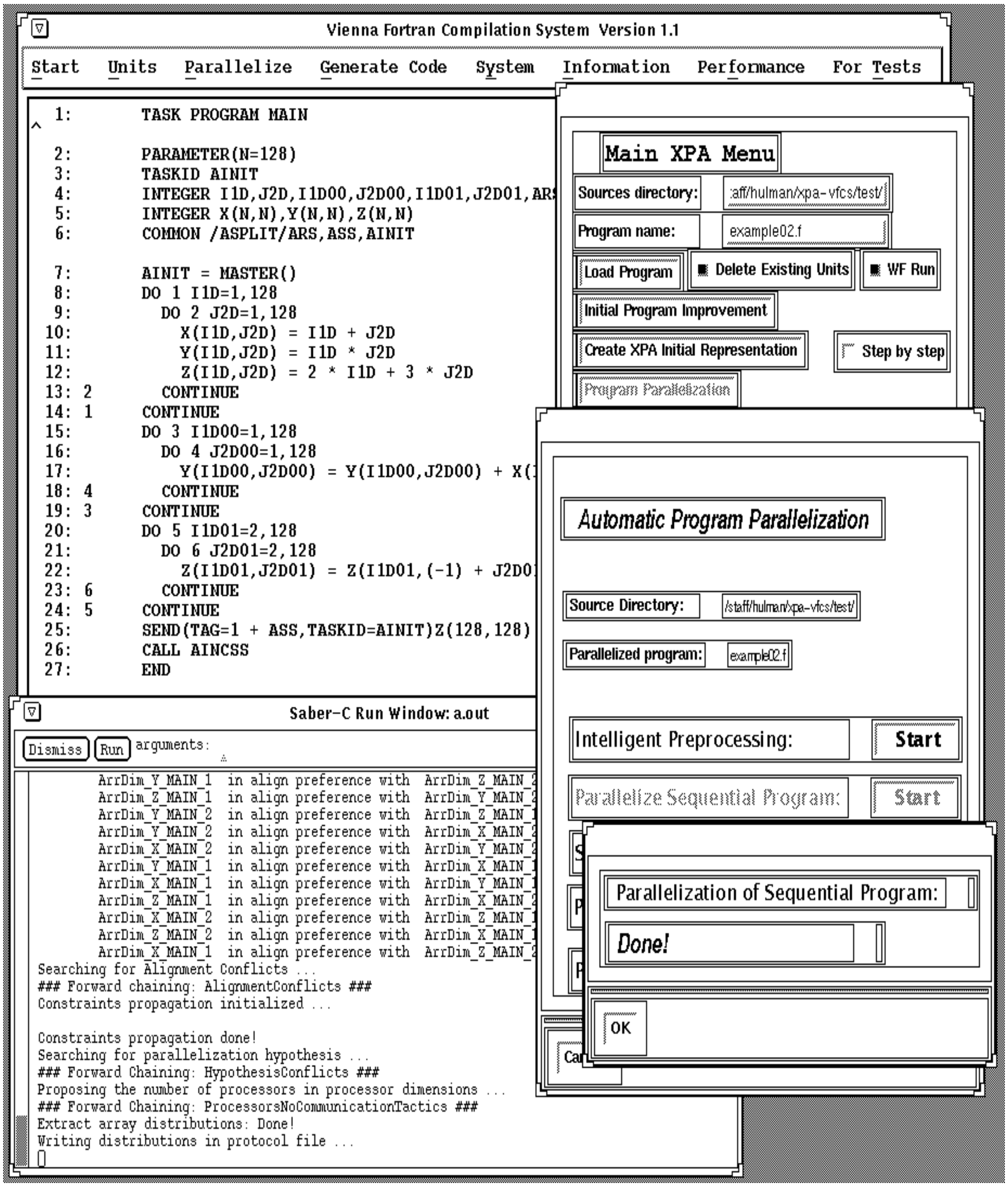

Figure 13: After Data Distribution Extraction

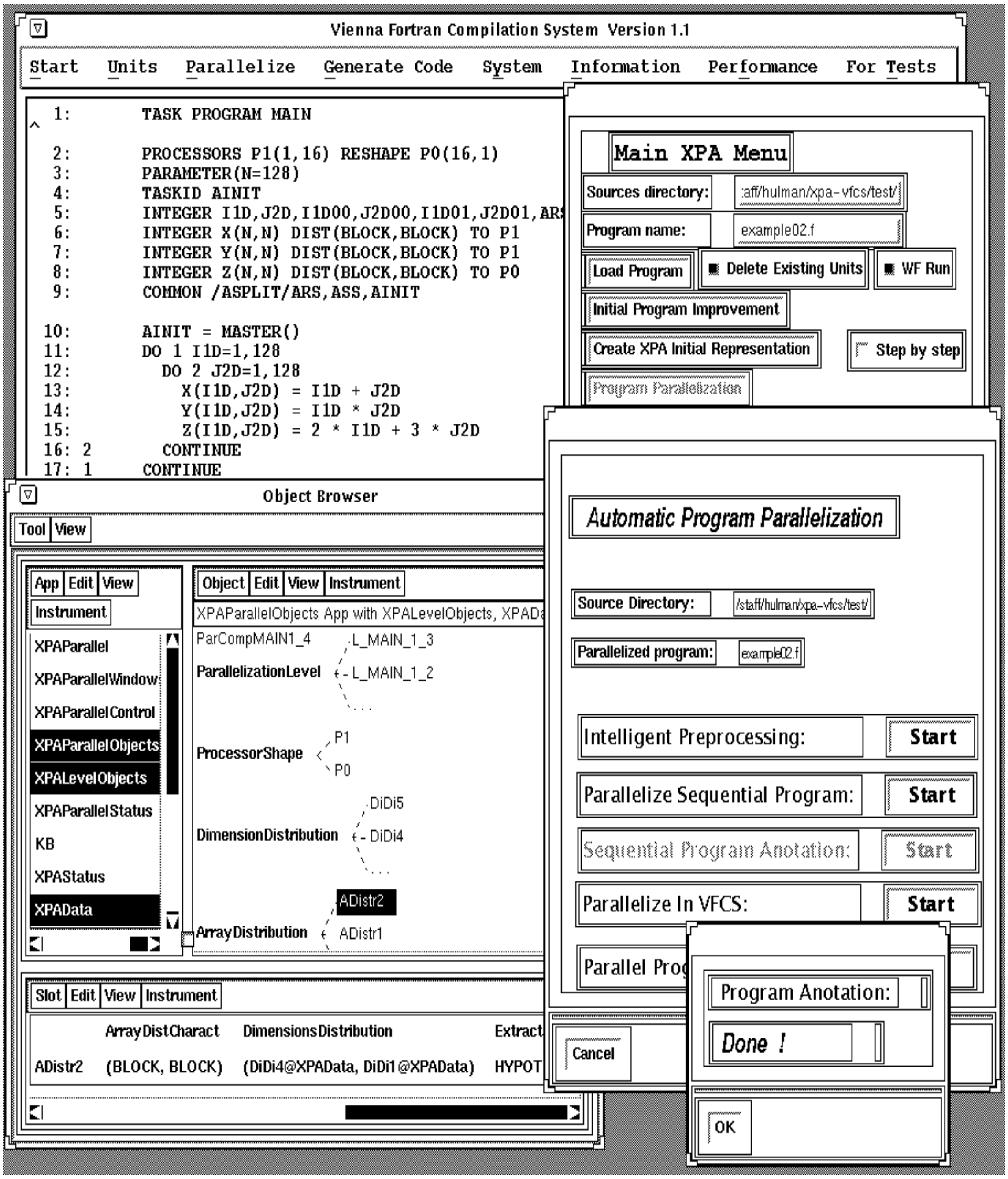

Figure 14: XPA after Sequential Program Annotation

in Section 6) and related work at the Universities of Indiana, Yale, Illinois, Rice, and Barcelona [22, 27, 16, 2, 9, 10].

In the future development of Expert Advisor towards a knowledge-based tool fully supporting parallel programs development a dual strategy will be employed:

- A set of **heuristics** will be developed, reflecting properties of applications, the programming language, the restructuring system, and the underlying target machine.
- An **analysis infrastructure** will be created. Based upon the reduced search space resulting from the application of heuristics to the program, the application of precise or approximate analysis becomes feasible. The resulting information can be used to control automatic compilation, or, at least, to provide the user with "intelligent" information on the major options for optimal program tuning.

The development of Expert Advisor is based on *expert system technology* whereby the complexity of the parallelization problem suggests an *incremental approach*: after each step the results are carefully evaluated to provide feedback for the next stages of development. We will start with the implementation of advanced *knowledge base clusters* and an associated *knowledge abstraction tool*, which provides support for the organization, storage and access to the domain specific knowledge now only available to experts in the field. Gradually we will improve the XPA towards a navigating system, which guides the programmer and compiler through the complex search space of optimizing transformations and data distributions. Moreover, the XPA will also include learning capabilities, such that the system automatically finds correct and performance efficient transformation sequences – based on an intensive training phase comprising many parallelization sessions of real codes – with a minimum of user interaction.

The development of Expert Advisor can be subdivided in two main steps.
The *first step* involves acquisition and representation of expert knowledge. It includes the development of a framework for **knowledge abstraction** and the selective integration of structured problem domain knowledge into **knowledge clusters**, which may cover (cf. Section 5.5):

- programming languages,
- parallelization paradigms,
- program transformations,
- data distribution strategies,
- compiler analysis information,
- performance parameters and performance tools,
- architecture models.

Knowledge about specific *application domains* can also be added to this list. The *program model* and *parallelization session model* may be considered also as separate knowledge clusters, being exceptionally important for representation of the achieved state during the program parallelization within Expert Advisor.

In addition to constructing the individual knowledge clusters, the relations between them and the knowledge itself must be modeled. *Knowledge abstraction* will be used to implement the different depths of knowledge important for specific tasks and users accessing the knowledge clusters. Through the use of knowledge abstraction and the clear definition of which view of knowledge is of importance in a specific situation it will be possible to integrate all the different knowledge clusters.

The *second step* involves the development a **navigation system** that provides the user with an *intelligent assistance* and *guidance* during the parallelization session. For instance, exploiting the knowledge in knowledge clusters, the navigation system will explain program transformations to the user, indicate if a specific code change is legal, and how the application of a transformation influences the performance parameters. Performance information will be made explicit by separately extracting and consequently displaying performance parameters, such as cache misses, communication and computation time, work distribution, and network contention. This information can then be related back to the original code, and thus enable the identification of problem areas. Based on the performance data characterizing the actual program state, the system will then provide the user with information regarding the choice of transformation strategies, depending on the parameters to be improved.

As a long term research topic, we intend to develop a powerful and intelligent parallelization system based on expert system technology, which targets **automatic compilation** for HPCs. This includes automatic selection of entire program transformation sequences, automatic static and dynamic data distribution, and intelligent detection of program patterns and their replacement by predefined efficient implementations.

If parallelization decisions cannot be made at compile-time, e.g. due to irregular problems, then we will investigate possibilities of using a knowledge based system for *dynamic decision making*.

Furthermore, we will develop *learning mechanisms* which will contribute to gradually automating many design decisions traditionally made by human experts.

7.2 Short Term Goals

In Sections 3 and 4 we outlined the required features of the Expert Advisor as we are implementing it within the VFCS parallelization environment. In Section 5 we proposed the XPA structure and its main modules. Section 6 presented the current state of the XPA implementation. In the next paragraphs, we will be more specific about the short term works extending the current XPA implementation towards the proposed version:

Program model extension. The current version of the program model, as has been implemented in the XPA (see Section 6.4), models features of sequential and data parallel programs. We intend to extend this model by new concepts, which will correspond to language constructs of message passing program, e.g. communication statement, mask of statement, etc. The performance aspects of a parallel program, as estimated by the P^3T, will be captured in the program model. The concept of *code pattern* will be inserted in the program model.

Session model development. We have recognized three phases of the program parallelization and performance tuning within the VFCS parallelization environment. As can be seen in Figure 2, realizing the second or the third phase of program tuning, the program is gradually transformed from the sequential form to the message passing form, and finally, to the target code. Usually, both phases are to perform program tuning in a cyclic and iterative manner. Therefore, the validity of the program model during performance tuning cycles has to be maintained. The session model must reflect all features of the parallelization session during tuning phases. A close interconnection between the program and session model is required.

Acquisition and representation of knowledge for code patterns. For acquiring knowledge related to code patterns we use a semi-automatic knowledge acquisition tool constructed according to ideas described in [25]. It performs the following two tasks: *measuring and collection of performance data*, and *graphical presentation of measured data*. After both tasks have been finished, the conclusions of the analysis of these graphs is formulated and represented in the form of forward-chaining and/or backward-chaining ProKappa rules.

Assistance and the guidance expertises for finding parallelization strategies induced by interpretation of the P^3T parameters. Automatic identification of hot spots in a program, based on investigation of actual program model after the P^3T execution, have to be implemented. Assistance and guidance expertise modes, which will support the application of program changes in order to tune performance, and which will take into account the parallel program parameters obtained from the P^3T, will be implemented. For instance, to improve the program's load balance, the data distribution has to be changed; to reduce the communication overhead, loop distribution, privatization of variables, and scalar expansion must be performed; by strip mining, loop interchanging and unrolling, the data locality can be improved.

Guidance in driving the parallelization session. Guidance will be offered regarding to selection of the next parallelization step, or regarding to choosing the most urgent subproblem to be solved.

New strategies for automatic data distribution. We intend to equip the Expert Advisor with a set of various *data distribution strategies*. These aim to help the user at gradual refinement and completion of the data distribution. Often, partial data distribution (not all arrays are distributed), will serve as the basis for subsequent extensions to get the final data distribution.

Automatic target code instrumentation. The indices, obtained by application of measuring during parallel program execution, give the most precise information about program performance. Usually, large amounts of performance data are collected after program execution. Different techniques and performance analysis tools must be applied in order to handle and analyze such data. Moreover, the behavior of the entire program as well as of its substructures must be analyzed. Therefore, selection of the program parts, which seem to be important to obtain an adequate information about performance of the program, is the key issue in instrumentation of the program. Selection of appropriate indices is also very important. In the Expert Advisor's program model, all parallelization problems, constraints, and applied program transformations are reflected. The critical points (program phases) are identified. This is a good basis for automatic instrumentation of the target code.

Integration of a new target architecture into intelligent parallelization. In the near future we plan to include two target architectures (Meiko CS-2 and Intel iPSC/860) in our prototype implementation. We have started with the Intel machine and will extend the knowledge base by knowledge related to the Meiko architecture.

7.3 Summary

We intend that the result of proposed work will be an advanced knowledge-based parallelization environment around VFCS offering the following main features:

- *intelligent support* for the VFCS parallelization session
- *integration* of VFCS compiler, parallelization support tools, parallelization paradigms
- *parallelization platform* for different target architectures
- *automation of the parallelization process* whenever possible
- *explicit representation and exploitation of expert knowledge*
- *incorporation of heuristics and ad-hoc solutions*
- *exploitation of the application domain dependent parallelization knowledge*
- *dynamic decision making* for run-time parallelization problems
- *learning facilities*

The Expert Advisor will be built using principles of modularity with the aim to allow:

- easy *integration* of new parallelization support tools
- rapid *adaptation* to new parallelization paradigms and target architectures
- easy *knowledge base enhancement* by new parallelization expert knowledge
- compiler *portability improvement*

8 Related Work

The application of an expert system technology for supporting parallelization for distributed-memory multiprocessor systems is an area of long term research. It has been topic of concern not only where DMMP's are involved, but also for vectorizing compilers. The pioneering work in this field was targeted to vectorization.

The expert advisor EAVE was developed to assist in the transformation of program for input to the IBM 3090 VF; it also looks for patterns within the code, and has rules telling it how code can be transformed to obtain an equivalent form that can be efficiently vectorized [5].

The experimental InParS system [3] implements heuristic state search algorithm A*. It is applied to automatically generate the sequence of transformations for optimal vectorization on IBM 3090/180S VF machine. A simple performance prediction is used to drive the search process.

The rule based transformation system ParTool [6] for semiautomatic SPMD code generation has been developed at the Delft University of Technology. ParTool enables the representation and reasoning about code transformations in the form of rewrite-rules. This system provides advanced pattern matching, but unfortunately lacks a flexible control mechanism for application of the rules.

Wang and Gannon [30] proposed an expert system for the hierarchical parallelization of programs to run on different multi-processors architectures. The proposal concentrates on the formal representation and specification of the knowledge about the transformation process. Wang has constructed an advanced knowledge acquisition tool in Prolog for collecting knowledge about various target machines architectures [31].

ATExpert [23], a tool developed at Cray Research, Inc., supports parallelization within the Autotasking compiling system. Based on run-time profile data, it graphically displays various aspects of parallel program performance. Expert systems are used to simulate parallel execution and help a user gain insights into understanding the parallel performance of a program.

The MPP Apprentice performance monitoring tool helps the user to find and correct performance anomalies and inefficiencies in programs that are designed for Cray MPP systems. The system provide a general advice on ways how to improve the code.

Parallel ELLPACK [18] is a domain-specific solver environment for partial differential equations. It emphasizes machine independence by knowledge-based mapping of user-specified computations to selected target architectures.

A proposal of the user friendly environment is presented in [11]. The Program Development Environment PDE, that is a part of the project, is a highly interactive and knowledge-based environment for parallel software development in selected domains of high-performance computing.

Kessler has developed a strategy for complete parallelization based almost entirely upon pattern recognition: the PARAMAT system attempts a recognition and algorithmic substitution for numerical patterns [21].

A lot of attempts have been made to develop techniques for automatic data distribution [19, 22, 27, 16, 2, 9, 10, 24].

9 Conclusion

The experimental system presented here is the basis for future development of the Expert Advisor for the VFCS parallelization session. The development effort to realize this experimental system resulted in the implementation of the low level kernel for the Expert Advisor. The VFCS-XPA interface will not require many extensions in the future. A sound basis has been created for preprocessing of XPA object base, the background for the knowledge base has also been created. Basic strategies for automatic data distribution have been proposed and implemented.

The experience with this experimental system have forced us to do a much deeper analysis to decide what parts of the system should be supported by the inference engine of the expert system. The active relations of the ProKappa expert system shell should be also used more frequently.

References

[1] Andel S., Hulman J.: “Automatic Data Partitioning within Vienna Fortran Compilation System”. Internal report of Institute for Software Technology and Parallel Systems, University Vienna. In preparation.

[2] Anderson J., Lam M.S.: “Global Optimizations for Parallelism and Locality on Scalable Parallel Machines”, SIGPLAN Programming Language Design and Implementation 1993, pp. 112-125

[3] Al-Ayyoub A., Yazici A.:“Knowledge-Based Program Parallelization”, Computer Science Department, Bahrain University, Isa Town, February 1992, 16 pp.

[4] Blasko R.: “Simulation Based Performance Prediction by PEPSY”, Proc. of the ”28th Annual Simulation Symposium”, April, 1995, Phoenix, USA, IEEE #95TH8049, pp. 341-349.

[5] Bose P.: “Interactive Program Improvement via EAVE: an Expert Advisor for Vectorization”, Proc. of the Int. Conf. on Supercomputing, St. Malo, July 1988, pp. 119-130.

[6] Breebaart L.C., Paalvast E.M., Sips H.J.: “A Rule Based Transformation System for Parallel Languages”, In: Proc. of Third Workshop on Compilers for Parallel Computers, Vienna, Austria, ACPC/TR 93-8, July 1992, pp. 13-21.

[7] Calzarossa M., Serazzi G.: ”Workload Characterization: A Survey”, Proc. of IEEE, 81(8): 1136-1150, August 1993.

[8] Chapman B., et al: “Vienna Fortran Compilation System. Version 1.0. User’s Guide”, Department of Statistics and Computer Science, University of Vienna, Austria, January 1993, 192 pp.

[9] Chatterjee S., Gilbert J.R., Schreiber R.: ”Mobile and Replicated Alignment of Arrays in Data-Parallel Programs”, Proc. Supercomputing ’93, Portland OR, November 1993, pp. 420-429

[10] Crooks P., Perrott R.H.: ”An Automatic Data Distribution Generator for Distributed Memory MIMD Machines”, Proc. Fourth Workshop on Compilers for Parallel Computers, Delft, The Netherlands, Dec. 1993, pp. 33-44

[11] Decker K.M., Dvorak J.J., Rehmann R.M., Ruhl R.: “Satifying Application User Requirements: A Next-Generation Tool Environment for Parallel Systems”, Pro. of Int. Conf. High Performance Computing and Networking, Milan, Italy, May 1995.

[12] Di Martino B.: “Design of a Pattern Matching Expert (PMX)”, Technical Report, Institute for Software Engineering and Parallel Systems, University of Vienna, April 1995.

[13] Fahringer T.: “The Weight Finder - An advanced Profiler for Fortran Programs”, Proc. of AP’93, Saarbrücken, Germany, 1993.

[14] Fahringer T.: ”Automatic Performance Prediction for Parallel programs on Massively Parallel Computers”, TR 93-3, Institute for Software Engineering and Parallel Systems, University of Vienna, September 1993.

[15] Gupta A., Prasad B.E. (eds.): “Principles of Expert Systems”, IEEE Press, 1988.

[16] Gupta M., Banerjee P.: “Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers”, IEEE Transactions on Parallel and Distributed Systems, 3(2), 179-193, March 1992.

[17] High Performance Fortran Forum: “High Performance Fortran Language Specification, Version 1.0”, Rice University, Houston, May 1993

[18] Houstis E.N., Rice J.R.: “Parallel ELLPACK: A Development and Problem Solving Environment for High-Performance Computing Machines”, In P.W. Gaffney and E.N. Houstis, editors, Programming Environments for High-Level Scientific Problem Solving, pp. 229-243, North-Holland, 1992

[19] Hudak D.E., Abraham S.G.: “Compiler Techniques for Data Partitioning of Sequentially Iterated Parallel Loops”, Proc. ACM Int. Conf. on Supercomputing, 1990, pp. 187-200

[20] Hulman J., Andel S., Chapman B., Zima H.: “Intelligent Parallelization within the Vienna Fortran Compilation System”, Proc. of Fourth Workshop on Compilers for Parallel Computers, December 1993, Delft, Holland.

[21] Kessler C.W.: “Automatische Parallelisierung numerischer Programme durch Mustererkennung”, PhD. Thesis, Saarbrücken University, Germany, 1994.

[22] Knobe K., Lukas J.D., Steele G.L.: “Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines”, Journal of Parallel and Distributed Computing 8, 1990, pp. 102-118

[23] Kohn J., Williams W.: “ATExpert”, Cray Research, Inc., Eagan, Minnesota, May 1992.

[24] Kremer U.: “NP-completeness of dynamic remapping”, Proc. Fourth Workshop on Compilers for Parallel Computers, Delft, The Netherlands, Dec. 1993, pp. 135-141

[25] Moritsch H.: “Knowledge Acquisition for Automatic Parallelization Based on Performance Test Generation and Evaluation”, Internal Report, Institute for Software Technology and Parallel Systems, University of Vienna, 1994.

[26] Pantano M., Blasko R., Moritsch H., Fahringer T.: ”VFCS-MEDEA Integration. Instrumentation. Parameters and Events Specification.”, Institute for Software Engineering and Parallel Systems, University of Vienna, Austria. Internal report, 1993.

[27] Ramanujam J., Sadayappan P.: ”Compile-Time Techniques for Data Distribution in Distributed Memory Machines”, TR-90-09-03, Dept. Electrical and Comp. Engineering, Louisiana State University, Sept. 1990

[28] “ProKappa User’s Guide. Version 2.0”. IntelliCorp, Inc., Publ. Number PK2.0-UG-2, October 1991.

[29] Rosti E., Serazzi G.: ”Workload Characterization for Performance Engineering of Parallel Applications”, IEEE, (9): 457-462, January 1994.

[30] Wang K.Y., Gannon D.: “Applying AI Techniques to Program Optimization for Parallel Computers”, Hwang. K., DeGroot D. (Eds.): Parallel Processing for Supercomputers and Artificial Intelligence, McGraw-Hill Pub. Company, 1989, Chap.12., pp.441-485.

[31] Wang K.Y.: “Intelligent Program Optimization and Parallelization for Parallel Computers”, Tech. Rept. No. CSD-TR-91-030, Department of Computer Science, Purdue University, April 1991.

[32] Wender B.: ”VFCS Instrumentation for Integration of MEDEA”, Technical Report, Institute for Software Engineering and Parallel Systems, University of Vienna, April 1994.

[33] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald: “Vienna Fortran – a language specification. Version 1.1”, ICASE Internal Report 21, ICASE, Hampton, VA, 1992.

[34] Zima H.P., Chapman B.: ”Supercompilers for Parallel and Vector Computers”, ACM Press Fronties Series, Addison-Wesley 1990.

[35] Zima H., Chapman B.: “Compiling for Distributed-Memory Systems”, Proc. of the IEEE, Special Section on Languages and Compilers for Parallel Machines, February 1993.