An Overview of the PIVOT Environment for Program Restructuring†

Chyi-Ren Dow Shi-Kuo Chang Mary Lou Soffa
dow@cs.pitt.edu chang@cs.pitt.edu soffa@cs.pitt.edu

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Abstract -- The objective of PIVOT, a program Parallelization and Visualization environment, described in this paper is to provide a programming environment that facilitates restructuring of programs for parallelization. The environment supports a range of facilities, including visual and textual forms of specifications for code transformations, automatic generation of transformers from specifications, interactive and undo facilities for code transformations, multi-paradigm program visualization, and multi-level browsing functions.

1. Introduction

Parallel architectures use various forms of parallelism to provide users with increased computational power. Programmers depend on parallelizing compilers to enhance or expose parallelism in the sequential code and to generate highly parallelized code. The basic problems when applying transformations include deciding which transformations to apply, the order in which to apply them, and the parts of a program that should be transformed. Another problem is that the application of a transformation changes code, making the original source code and the parallelized source code very different textually. The user may need to see the transformed code in order to determine the effects of the transformations on potential transformations in the code and, importantly, needs to be able to see the code that is actually executing to properly debug and test the program. These problems cause difficulties for the user when interacting with the parallelizing software or a debugger. The user needs system supports and guidance to facilitate the understanding and effective use of code transformations.

Many visualization techniques have been developed in recent years to aid in the programming and understanding of programs. Various visualization systems have been proposed and designed for sequential languages [6, 8, 9]. The importance of these visual systems is increasing with the recognition of the necessity of visual displays for understanding both the effects of transformations on parallelized program code and the parallel execution of such code. Although numerous parallelizing tools have been designed and implemented, such as PTRAN [1], ParaScope Editor [5], Parafrase II [7], SUPERB [13], and Mimdizer [10], some useful visualization techniques, such as effective ways of displaying transformed and untransformed code representations during parallelizing, are not developed or integrated into these systems. The systems are extremely limited and are based on a few selected transformations. No undo facility, a very important facility in interactive environments, is supported in these systems.

![Figure 1. Program visualization for parallelized code.](image-url)

We approach these problems by the design and development of PIVOT, a program Parallelization and Visualization environment. Program visualization can be regarded as a mapping from programs to graphical representations of the program [9]. PIVOT uses various program visualization and human-computer interaction techniques to facilitate the understanding and effective use of parallelization tools to expose parallelism in sequential programs. In our approach through PIVOT (Figure 1), a source program can be transformed into a graphical representation for visualization (2). This graphical representation can then be manipulated and
transformed into the graphical representation for the target program (4). In other words, instead of (or in addition to) performing direct transformations on the source program (1), the user has the option of visualizing the source program (2), performing transformations in the visual representation using visual transformation operators (4), and then transforming the resulting visual structure into the target program (8). Finally, by supporting the undo operator (5) (6) as well as mappings between the source and the graphical representations (7) (8), the user can try different alternatives and undo unpromising approaches. Thus, the user is allowed free movement among these paradigm representations.

In the remainder of the paper we describe the PIVOT environment, starting with an overview of the environment in the next section. Section 3 discusses the components of PIVOT. Section 4 describes the PIVOT prototype and conclusions are drawn in Section 5.

2. Overview of PIVOT

The objective of the PIVOT research project is to provide a programming environment that facilitates restructuring of programs for parallelization. The environment is composed of various software components, including program representations (source code, intermediate code, and target code), transformation specification languages, parallelizing tools (compiler frontend, transformer generator, etc.), visualization tools (multi-level browser, performance visualization tools, etc.), and other utilities (debugger, editors, etc.).

A sequential source program that is to be restructured for parallel execution is first translated to an intermediate program representation. PIVOT uses a transformation generator (PDG-Genesis) to generate transformers from transformation specifications. In addition to a textual transformation specification language (PDG-GOSpeL), an extension of GOSpeL [12] with PDG-based primitive actions, PIVOT also provides the user with a visual transformation specification language for the specification of transformations. To avoid having to specify commonly used transformations, a library of specifications (which can be tailored) and the generated transformers are part of the system. In order to support parallelization guidance, PIVOT provides a transformation catalog to help the user select the most appropriate transformations, given the desired goal [3].

This environment is targeted to application programmers as well as compiler writers. When restructuring a program, the user can either select a transformation to perform from the transformation library, tailor a transformation in the library, or specify a new transformation. With various visualization facilities supported by PIVOT, the user can see the effects of the transformation on the code and its behavior (e.g., performance visualization once the performance of the transformation is evaluated). The user may continue the parallelization process by selecting another transformation to perform or by removing a transformation from the applied transformation sequence without starting all over again. The user can also tailor new transformations in the PIVOT environment by changing a transformation specification to exploit more parallelism from the program. The user can express code transformations by using a visual specification language that eases the task of specifying transformations. With the PIVOT environment, the user can easily perform experiments to develop, analyze, and evaluate parallelizing transformations.

3. Components of PIVOT

This section describes the important components of PIVOT, including the program representations used by all tools of PIVOT, a visual transformation specification language, and the four components of PIVOT to specify transformations, generate transformations, apply transformations, and remove transformations.

3.1. The Program Representations in PIVOT

Due to the interactions between scalar optimization and parallelization, integrating scalar and parallelizing transformations can improve the use of parallelism and memory hierarchy [11]. Our two-level representation integrates two program representations, a high level one - VPDG [2], and a low-level one - DAG, to allow the application of both parallelizing transformations and traditional optimizations. VPDG, an extended form of the PDG, not only enables the exploitation of parallelism in sequential programs by applying transformations but also facilitates mappings for code visualization. Advantages of using the two-level representations include

1) Optimizing and parallelizing transformations can be freely intermixed.
2) Transformations can use both high/ low level information (e.g., data dependence/ data flow).
3) Code generation and code scheduling for architectures with different granularity can be supported by the two-level representation.
4) With appropriate information annotated on the representation [4], an undo facility for both optimizing and parallelizing transformations can be supported.

3.2. VOSpeL

With the increasing interest of using graphical intermediate representations for parallelizing programs, the visual specification approach thus becomes important.
A Visual Optimization Specification Language, VOSpeL, provides the user with a uniform model to visually specify and perform code transformations [3]. The VOSpeL model uses visual representations to express specifications of transformations and is based primarily on the PDG model.

A VOSpeL structure is a triple, \( G = < N, E, \phi > \) in which the definitions of \( N \) and \( E \) are the same as the node and edge sets in the PDG. \( \phi \) is a mapping from the sub-graph of \( (N, E) \) to annotation expressions. The basic node types include statement, predicate, and region nodes. The basic dependences include control dependences, flow dependences, anti-dependences, and output dependences. The combination of nodes and control dependences represents the control dependence graph of a code pattern involved in the optimization. Augmented with data dependences and appropriate annotation symbols, the valid conditions for applying an optimization can be specified. The annotation symbols include quantifiers, relation operators, logical operators, and some special symbols. After specifying the pattern of code and the needed dependence conditions of a transformation, the user then needs to specify the actions of applying a transformation. Five primitive actions, Add, Copy, Delete, Modify, and Move, are supported.

3.3. Transformation Generation

A PDG-based optimizer generator (PDG-Genesis) has been developed to support the automatic generation of code transformations from transformation specifications. The choices in implementing a PDG-based transformation generator include (1) separate and different transformation generators for visual and textual specifications, and (2) a uniform transformation generator for both specifications with the aid of a translator from visual to textual specifications, or vice versa. In order to enable the user to use VOSpeL, PDG-GOSpeL, and a mixture of two, and to reduce the efforts for the implementation of the two separate transformation generators, the second approach is used in the implementation of PDG-Genesis in PIVOT. Since the Genesis system [12] that automatically produces optimizers from GOSpeL specifications was already developed and implemented, PDG-Genesis, a PDG-based transformation generator, is revised from the available code of Genesis with the aid of a translator from VOSpeL to PDG-GOSpeL [3].

3.4. Transformation Application

The user of PIVOT can employ any code transformation that can be specified rather than relying on particular implemented transformations. The user can either select a transformation to perform from the transformation library or specify a new transformation. When restructuring a program, the user may need guidance to determine which transformation to apply. Various transformation guides and visualization facilities are provided by PIVOT to enable the user to easily and efficiently select a transformation to apply during the restructuring process [3]. One is to provide a transformation catalog by using a heuristic that collects only appropriate transformations from the transformation library. Another solution is to provide various visualization facilities to enable the user to see the effects of the transformation on the code and its performance evaluation. Applicable transformations are highlighted in the transformation catalog to assist the user in selecting a transformation to apply. The user can browse the applicable transformations in the catalog to have the corresponding code in the program views highlighted. The user can also browse a program representation to have applicable transformations in an application point of the program (e.g., a loop) highlighted. After the application of a transformation, the user may need to see the restructuring code as well as the effects of the transformation on potential transformations in the code.

3.5. Transformation Removal

An undo feature is provided in PIVOT which allows users to reverse applied code transformations. After the application of code transformations, the user can undo them in the reverse application order. The user may also undo transformations in an order independent of application order [4]. The technique uses post conditions of a transformation to determine whether the transformation can be immediately removed. The technique employs inverse primitive actions, making it transformation independent. The enabling and disabling interactions of transformations are used to drive the process, thereby reducing redundant analysis when undoing transformations.

4. The PIVOT Prototype

This section presents the PIVOT prototype. The functionalities of the prototype are discussed first followed by a program parallelization example.

4.1. Functionalities of the Prototype

A snapshot of the PIVOT prototype is shown in Figure 2. A pulldown menu bar appears at the top of the figure. The first four buttons are pulldown menus and the Help button is a push button used to invoke online help for the system. The following summarizes the kinds of operations found in each menu:
• File Menu: Operations to load and save files and to quit the system.
• Browser: Operations to enable/disable the browsing function and control the browsing level.
• Auxiliary: Operations to present a system demonstration and invoke various integrated tools such as the PDG C compiler (PDGCC), performance visualization tools, debuggers, and graph pattern matching tools.

As shown in the middle part of Figure 2, there are three windows for the source code, the transformed source code (target), and the program dependence graph. The Opt Catalog is located at the lower right part of the figure and applicable transformations can be highlighted by the system to assist the user in performing legal transformations. The Transformation Sequence window located at the lower left part of the figure is used to record the applied code transformations and maintain them in a transformation list. Two push buttons to the right of the transformation sequence are used to undo code transformations. One is "undo in order" to remove transformations in the reverse application order. The other is "undo in any order" to remove transformations in an order independent of application order.

4.2. A Program Restructuring Example

When restructuring a program, the user can either select a transformation to perform from the transformation library, tailor a transformation in the library, or specify a new transformation. As shown in Figures 2 - 3, the following scenario illustrates a session of restructuring programs using PIVOT. This scenario consists of two important components of PIVOT to apply transformations and remove transformations. Examples to specify transformations by the VOSpeL editor and to generate transformations using PDG-Genesis are described in [3].

(1) Applying Transformations: After the user opens an existing source file and invokes PDGCC to translate a C program to a program dependence graph, applicable transformations are highlighted in the Opt Catalog by the system to assist the user in applying transformations. The user can browse the highlighted transformations in the transformation catalog to have the corresponding code in
As shown in Figure 3, the user has applied transformations in the sequence of DCE, INX, ICM, and FUS where ICM was enabled by INX. The applied transformation is recorded in the Transformation Sequence Window and the applicable transformations are updated in the Opt catalog. The window at the lower left part of the figure contains the transformation sequence interactively applied by the user. Using the multi-level browser, the user can browse a single statement or a block of statements in one program representation and the corresponding code of other views of the program and the corresponding transformations in the transformation sequence will be highlighted. The user can also browse the transformation sequence to have the corresponding code in the program views highlighted. The simultaneous visualization of different program representations of a program as well as the applied transformations helps the user in understanding code transformations and in debugging of transformed code, and allow the user to assist in parallelizing the program interactively.

(2) Removing Transformations: As shown in Figure 3, two push buttons are used to undo code transformations. One is undo in order. The other is undo in any order (arbitrary undo). Regarding to undo in order, the first time the undo command is issued, the last transformation is undone. Consecutive repetitions of the undo command continue to reverse earlier transformations. The technique has been implemented in this research. If the user clicks the undo-in-order button, the loop fusion transformation in Figure 3 will be reversed and the FUS button will be highlighted by a different color to denote it has been undone and is still applicable [3].
5. Conclusions

This paper describes the PIVOT environment to facilitate the understanding and effective use of parallelization tools to expose parallelism in sequential programs. Various visualization, parallelization, and human-computer interaction techniques are employed in the PIVOT environment. The values of PIVOT to users are (1) specified transformations rather than specific implemented transformations, (2) a visual transformation specification model that eases the task of specifying transformations, (3) users see the effect of transformations on target code, (4) for any inappropriate transformation, the user can delete it without starting all over again, and (5) ease of performing experiments on transformations.

One important application of the PIVOT environment is to perform experiments to develop, analyze, and evaluate parallelizing transformations. One advantage of using PIVOT is a better understanding of program optimizing and parallelizing transformations with various visualization facilities supported by PIVOT. The user can improve his/her ability to understand the process of restructuring programs from the graphical display of both untransformed and transformed program code. Another application is in symbolic debugging of parallel programs. The visual interfaces to debuggers can be explored for static program transformations or dynamic program executions. Another application is in Computer Aided Instruction (CAI). Users can learn some program development processes such as transformations from graphical feedback and interactive communication with the computer. Some principles and techniques of courses in programming language and compiler design can be taught in the lab using the program visualization environment.

Future directions include the development of a full implementation of PIVOT. Experiments should be conducted to measure the performance of techniques for undoing transformations. We also intend to develop and incorporate into PIVOT a performance visualization tool that supports various performance measurements, provides multiple display paradigms, and maintains mappings between program constructs and performance visualization items. Next, we will investigate techniques to debug parallelized code in PIVOT. Various visual aids for debugging parallel code will be developed. The final step will be to investigate techniques to enable the user to incrementally transform parallelized programs through the PIVOT environment.

References