Symbol Table Management in an HPF Debugger

Marian Bubak[1,2], Włodzimierz Funika[1], Grzegorz Młynarczyk[1], Krzysztof Sowa[3], Roland Wismüller[4]

[1] Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059, Kraków, Poland
[2] Academic Computer Centre – CYFRONET, Nawojki 11, 30-950 Kraków, Poland
[3] Institute for Software Technology and Parallel Systems, University of Vienna, Liechtensteinstrasse 22, A-1090 Vienna, Austria
[4] LRR-TUM, Technische Universität München, D-80290 München, Germany
{bubak,funika,mlynar}@uci.agh.edu.pl, sowa@par.univie.ac.at, wismuell@in.tum.de
phone: (+48 12) 617 39 64, *fax:* (+48 12) 633 80 54

Abstract. In this paper we present a functionality and a structure of a symbol table manager for debugging HPF programs. The manager retrieves information from executable files generated by different Fortran 77/90 and HPF compilers and converts it into an internal representation independent of file format, which is accessed by the debugging system.

1 Introduction

Debuggers are a critical part of the software engineering process. They allow to control execution of an application and to test its correctness. Most of present debuggers are capable to control only one single process at a time. Though there are also some parallel debuggers, they only support explicit message passing programs and are not sufficient for HPF applications. Programs written in HPF are translated by compilers to run as a set of intercommunicating processes. For rapid, effective development it is important to have a possibility to observe behavior and to track bugs in application interactively at the HPF source code level while leaving the handling of all parallelisation issues to the debugger.

Fig. 1 presents an overview of the *DeHiFo* debugger that is being developed to address source level debugging of HPF programs [2]. The debugging system is dedicated to support programs compiled with the Vienna Fortran Compiler (VFC)[1] which is a source-to-source compilation system translating HPF applications to Fortran 90 (F90) programs. To generate an executable file, a vendor back-end F90 compiler is used. The two-stage compilation process is reflected in the debugger architecture. The main parts of the system are the *Base Debugging Subsystem (BDS)* and the *HPF Dependent Debugging Subsystem (HPF-DS)*. BDS operates as a low level debugger closely related to the machine which it is running on. It should resolve all platform specific issues and hide them from the HPF-DS level. It must also define a clear, simple but unequivocal interface that provides functionality allowing to inspect the state of processes and values of data in the parallel program. When executing, this layer does not consider the consistency of the running application with the HPF source code but provides information on each process of the program. The design relies on the *DETOP* parallel debugger [3] and the *OCM* monitoring system developed at LRR-TUM. HPF-DS works on top of the BDS and provides a higher level functionality which includes the source level (HPF) view of the parallel program execution, interactive control and altering the application data.

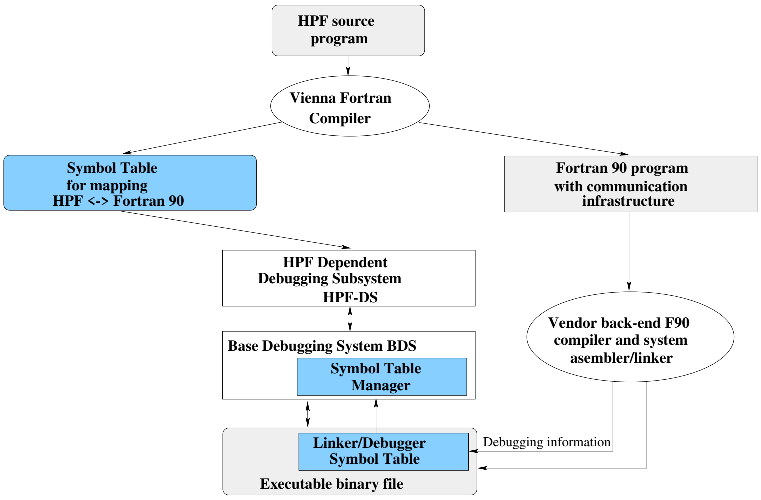

Fig. 1. Overview of the HPF debugger architecture

2 Extracting Information from Symbol Tables

In a source level debugger, the user operates on the application's objects using only symbol names (e.g variables and procedures names). The debugging system requires two mappings: *target-to-source* (F90 $\rightarrow$ HPF at HPF-DS level, machine addresses $\rightarrow$ F90 at BDS level) and *source-to-target* (HPF $\rightarrow$ F90 at HPF-DS level, F90 $\rightarrow$ machine addresses at BDS level). Whenever a break occurs in the target code, the original statements related to the current position can be located in the source program using the first mapping type. The latter one, in turn, is used by debuggers for setting breakpoints and obtaining information about the values of variables.

Before the program starts executing, the debugger has to recognize the structure of the program being debugged based on the HPF symbol table file and information stored in the executable binary file. When compilation is performed with debugging options, the HPF compiler produces a symbol table file which contains the relations between HPF and F90 programs and the F90 back-end compiler generates a special symbol table that describes the structure of the source program. Since HPF is a set of directive-based extensions for Fortan 90, designed to allow specification of data parallel algorithms, the scoping structure of the source and target programs is the same. Therefore to avoid redundancy the symbol table file at the HPF level contains information only about HPF specific symbols, e.g., distributed data, processor arrays, and program transformations, i.e., line mapping information, effects of program parallelization and compiler optimizations. This information is used to provide single data and control flow views of the parallel program and to handle compiler optimizations like data value problems, i.e., detecting whether at the given point in the program the value of variable can be inconsistent with the user's expectations, and finding out the current value of the variable.

Since the requirements for the source-to-source symbol table do not closely match existing formats we decided to specify a new format in ASCII to provide portability. Symbol table information is generated for each HPF source file that is processed and stored in a separate file that can be left alone or attached to an object file. Information on a single HPF source file is enclosed in a section containing a sequence of records with a description of variables with HPF attributes or affected by compiler transformations, as well as descriptions of parallel constructs in a program, e.g., INDEPENDENT loops. These records are enclosed in blocks indicating their scope. Additionally there are line translation records which provide a line number table to map between HPF source line numbers and Fortran 90 source line numbers.

After compilation, the linker puts together the object files of different compilation units, combining also their debugger symbol tables. There are several file formats, e.g., ELF, COFF, XCOFF, a.out. Moreover, in the same format there is a variety of extensions in the information structure generated by different compilers. On some platforms, the linker fails to update the address-related information in the symbol tables. Thus, debuggers have to obtain this data from other parts of the executable (related to linker symbols) which are specific to the particular file format.

In order to keep the debugging system portable, a symbol table management library *ST_LIB* has been developed [3]. In this work, we have extended the functionality of this library to support F90/HPF compilers. It is able to read files in various formats (executable, ASCII) and convert the debugging information to the format- and source language-independent, internal structure. The library can manage several symbol tables at the same time, which allows the debugger to handle programs that are executed on heterogeneous clusters as well as programs that consist of different binary files dynamically loaded during execution. symbol table

At the very beginning of the debugging process the executable file is opened by a common reader, which determines the target machine and the file format used to build the file. Next, the format specific reader is called to transform all relevant data to the internal structure. The debugging information is structured according to the program's organization. This structure also influences the internal debugger representation which includes the description of the main program objects: source files associated with the program, procedures (subroutines), functions, variables, data types, mapping between source and target statements.

The internal representation is a tree-like structure. At every level the objects contain not only information specific to them but also the symbol scope and references to all subobjects defined in their range. This is crucial for controlling the program execution. Each statement in the program is associated with its particular scope, and indirectly with the other scope containing that scope. When a user wants to choose a symbol, this is always done using the symbol's name. In this case the debugger has to discover which symbol the name refers to. This process is known as a symbol lookup and is always performed with respect to the current object scope. The interface to access the symbol table's internal representation behaves like a pattern matcher but its matching features are more sophisticated than only mapping names to symbols.

Since for each object the symbol table manager stores a complete specification describing this object, it is capable to return a set of objects that fulfil the given criteria. Apart from peculiarities, all symbol specifications have a common part which consists of symbol name and address, symbol class (e.g. variable, procedure, type definition, constant) and subclass which determines the primary scope of the object (e.g. local/global symbol, function parameter).

It is possible to make requests with various conditions. The system supports wildcards to substitute for each component in the symbol specification. Different search types allow to find all descendant symbols that are, for instance, only local to the current scope or global to the whole program. In this way the symbol manager provides an interface which supports different executable formats, but also allows the debugging system to retrieve information about a specific symbol and to map it backwards and forwards between source and target levels. The manager architecture supports search for a set of symbols with a given class (and/or subclass) which enables to obtain information in the debugging process, for example, about all variables defined in the specified scope.

Acknowledgement. We would like to thank Prof. Peter Brezany for valuable discussions. This research was supported by the Austrian-Polish grant.

References

1. Benkner, S., Sanjari, K., Sipkova, V. and Velkov, B.: Parallelizing Irregular Applications with the Vienna HPF+ Compiler VFC. In: Proc. Int. Conf. High Performance Computing and Networking, vol. 1401 of Lecture Notes in Computer Science, 816-827, Amsterdam, The Netherlands, April 1998. Springer.
2. Brezany, P., Grabner, S., Sowa, K., Wismüller, R.: DeHiFo - An Advance HPF Debugging System. In: Proc. of the 7th Euromicro Workshop on Parallel and Distributed Processing, PDP'99, 226-232, Madeira, Portugal, 3-5 February 1999.
3. Oberhüber, M., Wismüller, R.: DETOP - An Interactive Debugger for PowerPC Based Multicomputers. In: Parallel Programming and Applications, 170–183, IOS Press, Amsterdam, May 1995. LRR-TUM, Germany.
4. Rosenberg, J.B.: How Debuggers Work: Algorithms, Data Structures, and Architecture. John Wiley & Sons, Inc., 1996.