1. Preface

The objective of program analysis is to automatically determine the properties of a program. Tools of software development, such as compilers, performance estimators, debuggers, reverse-engineering tools, program verification/testing/proving systems, program comprehension systems, and program specialization tools are largely dependent on program analysis. Advanced program analysis can: help to find program errors; detect and tune performance-critical code regions; ensure assumed constraints on data are not violated; tailor a generic program to suit a specific application; reverse-engineer software modules, etc. A prominent program analysis technique is symbolic analysis, which has attracted substantial attention for many years as it is not dependent on executing a program to examine the semantics of a program, and it can yield very elegant formulations of many analyses. Moreover, the complexity of symbolic analysis can be largely independent of the input data size of a program and of the size of the machine on which the program is being executed.

In this book we present novel symbolic control and data flow representation techniques as well as symbolic techniques and algorithms to analyze and optimize programs. Program contexts which define a new symbolic description of program semantics for control and data flow analysis are at the center of our approach. We have solved a number of problems encountered in program analysis by using program contexts. Our solution methods are efficient, versatile, unified, and more general (they cope with regular and irregular codes) than most existing methods.

Many symbolic analyses are based on abstract interpretation, which defines an abstraction of the semantics of program constructs in a given language, and an abstraction of program input data. The abstraction of data and programs leads to a less complex view which simplifies many program analyses. Some problems which were originally undecidable become decidable or can be solved only with a very high computational effort. On the other hand, information can get lost through abstraction; thus a solution of a problem based on abstract interpretation may no longer be a solution of the original problem. The art of symbolic program analysis is focused on the development of a suitable abstract interpretation for a given program and its input data which allows the problems of interest to be solved.

Previous approaches on symbolic program analysis frequently have several drawbacks associated with them:

- restricted program models (commonly exclude procedures, complex branching and arrays),
- analysis that covers only linear symbolic expressions,
- insufficient simplification techniques,
- memory- and execution-time-intensive algorithms,
- either control flow or data flow can be modeled but not both,
- unstructured, redundant and inaccurate analysis information, and complex extraction of analysis information,
- additional analysis is required to make the important relationship between problem size and analysis result explicit,
- recurrences are frequently not supported at all, or separate analysis and data structures are required to extract, represent, and resolve recurrence systems.

Furthermore, for program analyses there is commonly a need to express values of variables and expressions as well as the (path) condition under which control flow reaches a program statement. Some approaches do not consider path conditions for their analysis. Without path conditions the analysis accuracy may be substantially reduced. We are not aware of any approach that combines all important information about variable values, constraints between them, and path conditions in a unified and compact data representation. Moreover, it has been realized [98] that systems with interfaces for off-the-shelf software are critical for future research tool and compiler development. Approaches that rely on specialized representations for their analysis are often forced to reimplement standard symbolic manipulation techniques which otherwise could be taken from readily available software packages. Symbolic expressions are used to describe the computations as algebraic formulas over a program's problem size. However, symbolic expressions and recurrences require aggressive simplification techniques to keep the program contexts in a compact form. Hence, *computer algebra systems* (CASs) play an important role in manipulating symbolic expressions and finding solutions to certain problems encountered in program analysis. During the last decade CASs have become an important computational tool. General purpose CASs [70], which are designed to solve a wide variety of mathematical problems, have gained special prominence. The major general purpose CASs such as Axiom, Derive, Macsyma, Maple, Mathematica, MuPAD, and Reduce, have significantly profited from the increased power of computers, and are mature enough to be deployed in the field of program analysis. One of the goals of this book is to introduce techniques for closing the gap between program development frameworks and CASs. Symbolic algebraic techniques have emerged in the field of CASs to broaden the spectrum of program analyses for compilers and program development tools.

We describe a novel and unified program analysis framework for sequential, parallel, and distributed architectures which is based on symbolic evaluation, and combines both data and control flow analysis to overcome or at least to simplify many of the deficiencies of existing symbolic program analysis efforts, as mentioned above. At the center of our framework is a new representation of analysis information, the *program context*, which includes the following three components:

- variable values,
- assumptions about and constraints between variable values, and
- conditions under which control flow reaches a program statement.

A key advantage of our approach is that every component of program contexts can be separately accessed at well-defined program points without additional analysis. We describe an algorithm that can generate all program contexts by a single traversal of the input program. Program contexts are specified as n-order logic formulas, which is a general representation that enables us to use off-the-shelf software for standard symbolic manipulation techniques. Computations are represented as symbolic expressions defined over the program's problem size. Instead of renaming data objects, our symbolic analyses try to maintain the critical relationship between a program's problem size and the resulting analysis information. This relationship is important for performance-driven program optimization. Our symbolic analysis framework accurately models assignment and input/output statements, branches, loops, recurrences, arrays (including indirect accesses), and procedures. Recurrences are detected, closed forms are computed where possible, and the result can be directly retrieved from well-defined program points. Note that detecting recurrences and finding closed forms for recurrences are decoupled. The decoupling simplifies the extension of our recurrence solver with new recurrence classes. All of our techniques target both linear and nonlinear symbolic expressions and constraints.

We intensively manipulate and simplify symbolic expressions and constraints based on a system which we have built on top of existing software. We will describe a variety of new techniques for simplifying program contexts. Furthermore, we have developed several novel algorithms for comparing symbolic expressions, computing lower and upper bounds of symbolic expressions, counting the number of solutions to a system of constraints, and simplifying systems of constraints. While previous work mostly concentrated on symbolic analysis for shared memory architectures, this research also supports symbolic compiler analysis for distributed memory architectures covering symbolic dependence testing, array privatizing, message vectorization and coalescing, communication aggregation, and latency hiding.

We do not know of any other system that models a similar large class of program constructs based on a comprehensive and unified analysis representation (program context) that explicitly captures exact information about

variable values, assumptions about and constraints between variable values, path conditions, and recurrence systems, and side-effect information about procedure calls.

Empirical results based on a variety of Fortran benchmark codes demonstrate the need for advanced analysis to handle nonlinear and indirect array references, procedure calls, and multiple-exit loops. Furthermore, we will show the effectiveness of our approach for a variety of examples including program verification, dependence analysis, array privatization, communication vectorization, and elimination of redundant communication.

Although symbolic program analysis has been applied to a variety of areas including sequential program analysis, parallel and distributed systems, real-time systems, etc., in this book we are mostly concerned with sequential, parallel, and distributed program analysis.

We have implemented a prototype of our symbolic analysis framework which is used as part of the Vienna High Performance Compiler (VFC, a parallelizing compiler) and P^3T (a performance estimator) to parallelize, optimize, and predict the performance of programs for parallel and distributed architectures. Although we examined our framework for Fortran programs on sequential, parallel, and distributed architectures, the underlying techniques are equally applicable to any similar imperative programming language.