DETERMINING COMPUTER ARCHITECTURE AND COMPLIER STRUCTURES THROUGH INDUCTIVE INFERENCE: A PRELIMINARY INVESTIGATION

by

M. Stauffer
R. S. Michalski

Report #906, Department of Computer Science, University of Illinois, Urbana, Illinois, January, 1983.

83-13

ORIGINAL

Determining Computer Architectures and Compiler Structures Through Inductive Inference: A Preliminary Investigation

M. D. Stauffer
R. S. Michalski

January 1983

File No. UIUCDCS-F-83-906

ISG 83-~~6~~3

January 1983

Determining Computer Architectures and Compiler Structures Through Inductive Inference: A Preliminary Investigation

M. D. Stauffer
R. S. Michalski
Department of Computer Science
University of Illinois
Urbana, Illinois

ISG 83-6

File No. UIUCDCS-F-83-906

ABSTRACT

With the current proliferation of computer architectures and programming languages, the selection of an appropriate machine-architecture and compiler structure has become a complex task, requiring special expertise. This paper presents a formulation of this task as an inductive learning problem and discusses some preliminary results.

Index terms: Knowledge Acquisition, Computer Architectures, Concept Learning, Inductive Inference, Decision Rules, Conceptual Data Analysis.

This work was supported in part by the National Science Foundation Grants No. MCS 82-05896 and MCS 82-05166.

Determining Computer Architectures and Compiler Structures Through Inductive Inference: A Preliminary Investigation

M. D. Stauffer
R. S. Michalski
Department of Computer Science
University of Illinois
Urbana, Illinois

1. Introduction

Designing a computer architecture and compiler structure that is well-suited to a given class of applications is a difficult problem, requiring special expertise and experience from a designer. The main difficulty is that the number of combinations into which the available hardware components and compiler modules can be arranged is potentially very large. Recently, a lot has been learned about this problem by the development of the PARAFRASE system [2]. This system is an optimizing preprocessor that determines data dependencies in a serial FORTRAN program and attempts to restructure the program in order to achieve the maximum speed-up on a parallel machine. In the process, PARAFRASE generates a large number of statistics characterizing the input program and the transformed program. The value determined for each statistic depends on the program and on the optimization steps required for the target machine architecture. At the present time, there are three main hypothetical architectures for which programs have been analyzed: SEA--Single-Execution Array, MEA--Multiple-Execution Array, and MES--Multiple-Execution Scalar. Future plans include a number of other hypothetical and actual machines, such as the CRAY, CYBER 205, etc.

For each (program,target machine/compiler) pair, PARAFRASE generates approximately one thousand statistics which describe the original program (number of DO loops, loop-nesting-depth), the transformations which were performed (number of loops interchanged, number of IF-statements removed), and the transformed source (number of vector operations, number of linear recurrences). In addition, a measurement of the resulting speed-up of the program is recorded. All of this data is stored in a large database which will eventually hold information on fifty packages of 10-40 programs each, times ten or more machine-architecture/compiler combinations, times the 1000+ statistics, i.e., approximately 400,000 data items altogether.

Two uses are envisioned for this data. One, more practical, use is for recommending an appropriate computer architecture to a client interested in running well-defined software packages. This can be done by using analogy: that architecture is suggested that performs well with programs that are similar to those in the client's package. Another, more theoretical, use is to develop insights and better understanding of the relationships that exist between various program characteristics and speed-up achievable on a given computer architecture and compiler structure.

A major difficulty faced in either case is the sheer volume of the data available. Another difficulty is poor understanding of the relevance of various variables and statistics to the task. Some variables, such as the number of linear recurrences, are obviously relevant to the choice of a particular machine architecture for a given program package. Other variables--such as the number of program statements--seem to be of lesser relevance. There are a large number of variables whose degree of relevance is still unknown. Clearly, a method for mechanically determining the relevance of variables and/or their combinations is highly desirable. Because of the volume of the data and the lack of established procedures for determining relevance, an Artificial Intelligence approach to the problem is necessary. This paper makes an attempt to attack this problem by formulating it as an inductive-learning task, and solving it by applying already-developed inductive-learning methodologies. We have adapted for this purpose an inductive program we have developed, GEM, whose predecessor AQ11 has been very successful in solving problems in other domains [4] [8].

2. Inductive Tools

GEM [9] is the latest in a series of inductive-learning programs that we have developed and experimented with which use the A^q algorithm (others include ESEL, AQ7UNI, AQ7, and AQLISP) [7] [10] [5]. The A^q algorithm [6] creates descriptions of object classes from examples of objects belonging to those classes. Examples are formulated as lists of attribute-value pairs. The fundamental basis of the A^q algorithm is the use of the negative examples of each class (typically the events of all of the other classes) as constraints on the generalization of the description of the class. If unlimited resources of time and space are available, the A^q algorithm will generate the most general description of each class possible given the input incorrect
data. As unlimited resources are not available, user-specified cost functions are used to choose and keep only the "best" hypotheses sufficient to account for the events of the class. The most often used cost functions have the effect of minimizing the number of conjunctive statements in the output descriptions, and of minimizing the number of variables present in each of the statements.

Our goals for GEM are:

1. To implement the A^q algorithm in a modularized, well-structured, and well-documented program suitable both as a "production" tool and as a basis for further research and experimentation.

2. To provide relational-table input and output compatible with the QUIN relational-database system [1].

3. To add constructive-induction--automatic creation of new variables which are logical or arithmetic functions of the existing variables.

4. To add the capability to induce arithmetic relations between variables, _a la_ BACON [3].

In addition to these capabilities, related projects are now under development which will provide other features to GEM as pre- and post-processors. These projects include VARSEL, a program to select most-likely-useful variables when there are more of the original variables than GEM can handle, and REVAL, a program to test unknown events against GEM-output descriptions.

3. Formulation of the Problem as an Inductive Task

Determining relevance of the variables in the PARAFRASE data can be defined as a problem of inductively learning general descriptions from specific examples. There are two ways of formulating the corresponding inductive task. One is to divide the programs in the database into classes associated with those architectures for which they can achieve maximum speed-up. The GEM program can then be applied to determine a unifying discriminant description of each class in the context of the other classes. These descriptions would indicate which variables and in what form are relevant for achieving the highest speed-up for a given class of programs.

Another formulation is to take all programs analyzed for a particular architecture, divide them into several classes on the basis of speed-up, and induce discriminant descriptions of the classes in terms of the program statistics recorded in the database. If a particular statistic did not appear in the description of the good or bad performers, or if it appeared but its range of values in the description spanned almost all of the possible values, then that statistic could be presumed to be irrelevant.

In general, let e be a description of a program (an "event") in the form of a list of program characteristics and generated statistics. (Some exemplary variables of the over 1000 defined were given in Section 1.) Suppose that this program was optimized for machine architecture A, after which it was compiled and executed. Suppose further that this optimized program obtained a speed-up in the "good" sub-range.

The above can be expressed as a production rule:

$$e \quad ::> \quad A_g$$

stating that if a program has description e, then it will run well on architecture A. If the speed-up for program e were "bad", we would have:

$$e \quad ::> \quad A_b$$

If we have more than one event associated with "good" or "bad" performance, then we have sets of productions:

$\{ e_i \quad ::> \quad A_g \}$, where e_i is in the set of good performers, E_g
$\{ e_j \quad ::> \quad A_b \}$, where e_j is in the set of bad performers, E_b

In the terminology of inductive learning, E_g and E_b can be viewed as sets of "positive" and "negative" learning examples (i.e., examples of good and bad choices of programs to be optimized for computer architecture A). Since, in general, we consider not one but several computer architectures, a general formulation of the problem is:

<u>Given</u>: Sets of production rules for "positive" examples associated with each computer architecture:

$$\begin{array}{l} \{ e_{i1} \quad ::> \quad A_{1g} \} \\ \{ e_{j1} \quad ::> \quad A_{2g} \} \\ \quad\quad\quad \cdot \\ \quad\quad\quad \cdot \\ \{ e_{k1} \quad ::> \quad A_{mg} \} \end{array}$$

where m is the number of architectures considered, and sets of production rules for "negative" examples:

$$\begin{array}{l} \{ e_{i2} \quad ::> \quad A_{1b} \} \\ \{ e_{j2} \quad ::> \quad A_{2b} \} \\ \quad\quad\quad \cdot \\ \quad\quad\quad \cdot \\ \{ e_{k2} \quad ::> \quad A_{mb} \} \end{array}$$

<u>Determine</u>: A set of rules:

$$\begin{array}{l} R_1 \quad ::> \quad A_{1g} \\ R_2 \quad ::> \quad A_{2g} \\ \quad\quad \cdot \\ \quad\quad \cdot \\ R_m \quad ::> \quad A_{mg} \end{array}$$

where each R_i is a condition composed of some logical combination of the program characteristics and generated statistics which best describes the corresponding class of good-performance programs.

4. An Initial Experiment

The programs from the EISPACK package (a set of programs for analysis of eigenvalue problems) were divided into five groups (eleven programs each) ranked by the speed-up achieved on the SEA architecture (as estimated by PARAFRASE). GEM was then applied to determine a discriminant description for characterizing the class of very good performers in the context of the other classes. One of the eleven programs in the "very good" class is represented graphically in Figure 1.

(a)

	Number DO Loops		
	1	2	3
(a)	0	6	16

(b)

		Number DO Loops		
		1	2	3
Number Vectorized DO Loops	0	0	0	0
	1	0	6	4
	2		0	12
	3			0

Figure 1

Input: Description of a program after optimization

The vector in Figure 1(a) shows the number of assignment statements in the program that were originally enclosed, respectively, in one, two, and three nested DO loops. The matrix in Figure 1(b) represents the same program after optimization. The columns again represent the original number of DO loops in which the statements in the columns were nested; the rows represent the number of the enclosing loops which PARAFRASE was able to vectorize (transform to a form where successive iterations are executed simultaneously on different processors). In our example, the value 4 in column 3, row 1 of Figure 1(b) indicates that out of 16 statements contained in depth-3 DO-loops (i.e., nested in 3 loops), 4 have had one of their enclosing loops vectorized by PARAFRASE.

The more DO-loops that are vectorized, the better should be the speed-up of the program. Accordingly, one might expect that the description of the class of very-good performers would contain relatively high values for the variables corresponding to the cells on the diagonal of the matrix, and relatively low values for the variables corresponding to the top rank of cells.

```
                     Number DO Loops                Number DO Loops

                      1     2     3                  1     2     3
                    -----------------            ----------------
              0     | 0-8 |     |    |           |     |     |     |
                    |----------------|           |----------------|
Number        1     |  0  |     |    |           |     |     | 1-7 |
Vectorized          -----------------|     or    -----------------|
DO Loops      2           |1-58 |    |                 |     |5-16 |
                          -----------|                 -----------|
              3                 |    |                       |    |
                                ------                       ------
```

Figure 2

Output: Alternate hypotheses in the description of the class of very-good performing programs

The results of GEM can be represented graphically by a diagram comparable to that of Figure 1. The two matrices shown in Figure 2 represent the two conjunctive statements which GEM produced to describe the class of programs which experienced very good speed-up. The first statement could be paraphrased as:

"Given a program, if the number of statements nested in only one loop, which cannot be vectorized, is from 0 to 8, there are no statements nested in one loop which can be vectorized, and the number of statements nested in two loops, both of which can be vectorized, is from 1 to 58, then the program belongs to the class which can be sped-up very well for this architecture."

The value sub-ranges mentioned can be given meaningful names such as "very_low", "medium", and "high", to make their significance more obvious.

The empty cells indicate variables which are irrelevant for discriminating the class of very-good performing programs from the other classes of programs. The remaining variables confirm, at least in part, our expectation that programs which speed up very well have relatively low values in the top rank of variables and higher values towards the diagonal.

5. Critique and Proposed Further Research

The variables and their values that were originally present in the input data may not be the most useful form of the data for determining relevance. In our example, for instance, the number of statements in a cell varies in importance from column to column, depending on the total number of statements in each column, since speed-up really corresponds to the percentage of statements that have been moved out of their original loop position--i.e., that have moved down their respective column. Constructive induction (automated construction of new numeric variables which are mathematical or logical expressions involving the input variables) could be used to create new variables that are normalized--that is, the original variables divided by the sum of the assignment statements in each's column.

Such constructive induction can be handled two ways. A domain expert using a system such as GEM can guide the variable creation process by suggesting combinations of variables and operators likely to be relevant, and declaring other combinations off-limits. Alternately, generate-and-test heuristics such as the trend-detectors used in the BACON system [3] can be used to automatically create useful new variables. Current research is focusing on these and other ways to add constructive induction of numeric variables to the GEM implementation of the A^q algorithm.

The rather arbitrary division of the input examples into classes of differing speed-up suggests another research problem: to extend the concept of continous versus discrete variable types to apply to the concept of classes. In our problem, for example, a more appropriate approach than dividing the range of speed-up into discrete classes might be to define a single continuous range of speed-up--very good to very bad--upon which programs can be placed. The descriptions of classes would then give way to a test (or tests) to determine placement of an "unknown" program on the range relative to the known examples, perhaps by generating a predicted speed-up. (Note that this is not the same as mathematical regression in that numeric values are ordinal but not necessarily interval.)

Another current research problem is finding ways to identify and name important or recurring sub-expressions in descriptions, resulting in a hierarchy of descriptions, and re-evaluating the distinction between variable and value in such a hierarchy.

6. Acknowledgements

We would like to acknowledge the contributions of Paul O'Rorke to the design and coding of GEM, and of Professor D. Kuck to the formulation of the PARAFRASE problem. This work was supported in part by the National Science Foundation under grants No. MCS-82-05896 and MCS 82-05166.

REFERENCES

[1] Cheng, A., "QUIN--Query and Inferential Database Sublanguage," to appear as a Master's Thesis of the Department of Computer Science, University of Illinois, Urbana, Illinois.

[2] Kuck, D., et al, "The Structure of an Advanced Vectorizer for Pipelined processors," Proc. 4th Intl. Computer Software and Applications Conf., October, 1980.

[3] Langley, P., "Rediscovering Physics with BACON.3," Proc. 6th Intl. Joint Conf. on Artificial Intelligence, pp. 505-507, August, 1979.

[4] Michalski, R. S., and Chilausky, R. L., "Learning by Being Told and Learning from Examples: An Experimental Comparison of Two Methods of Knowledge Acquisition in the Context of Developing an Expert System for Soybean Disease Diagnosis," A Special Issue on Knowledge Acquisition and Induction, Policy Analysis and Information Systems, No. 2, 1980.

[5] Michalski, R. S., and Larson, J. B., "Selection of Most Representative Training Examples and Incremental Generation of VL_1 Hypotheses: The Underlying Methodology and the Description of Programs ESEL and AQ11," Report No. 78-867, Dept. of Computer Science, University of Illinois, Urbana, May, 1978.

[6] Michalski, R. S., "Synthesis of Optimal and Quasi-Optimal Variable-Valued Logic Formulas," Proceedings of the 1975 Intern. Symp. on Multiple-Valued Logic, Bloomington, Indiana, May 13-16, 1975.

[7] Michalski, R. S., "A System of Programs for Computer-Aided Induction: A Summary," Proc. 5th Intl. Joint Conf. on Artificial Intelligence, MIT, Boston, August, 1977.

[8] O'Rorke, P., "A Comparative Study of Two Inductive Learning Systems," Internal Report No. 82-1, Intelligent Systems Group, Dept. of Computer Science, University of Illinois, Urbana, Illinois, 1982.

[9] Stauffer, M. D., "GEM/0--User's Guide and Program Description," to appear as a report of the Department of Computer Science, University of Illinois, Urbana, Illinois.

[10] Stepp, R., "Learning Without Negative Examples via Variable-Valued Logic Characterizations: The Uniclass Inductive Program AQ7UNI," Report No. 79-982, Dept. of Computer Science, University of Illinois, Urbana, July, 1979.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-F-83-906	2.	3. Recipient's Accession No.
4. Title and Subtitle Determining Computer Architectures and Compiler Structures Through Inductive Inference: A Preliminary Investigation			5. Report Date January 1983
			6.
7. Author(s) M. D. Stauffer and R. S. Michalski			8. Performing Organization Rept. No.
9. Performing Organization Name and Address Department of Computer Science University of Illinois Urbana, IL			10. Project/Task/Work Unit No.
			11. Contract/Grant No. MCS 82-05896 MCS 82-05166
12. Sponsoring Organization Name and Address National Science Foundation Washington, DC			13. Type of Report & Period Covered
			14.

15. Supplementary Notes

16. Abstracts

With the current proliferation of computer architectures and programming languages, the selection of an appropriate machine-architecture and compiler structure has become a complex task, requiring special expertise. This paper presents a formuation of this task as an inductive learning problem and discusses some preliminary results.

17. Key Words and Document Analysis. 17a. Descriptors

Knowledge Acquisition
Computer Architectures
Concept Learning
Inductive Inference
Decision Rules
Conceptual Data Analysis

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 9
	20. Security Class (This Page UNCLASSIFIED	22. Price