A Practical Application of Sharing and Freeness Inference

M.J. García de la Banda, M. Hermenegildo

Universidad Politécnica de Madrid (UPM),
Facultad de Informática,
28660-Boadilla del Monte, Madrid - Spain
e-mail `maria,herme@fi.upm.es`

(Extended Abstract)

1. Introduction

Abstract Interpretation [6] of logic programs ([1], [14], [8],[19], [2], [4], [13], [5], [12], [15] ...) is currently proposed as a means for obtaining characteristics of the program at compile-time, thus allowing several types of optimizations. However, only few studies have been reported analyzing the practicality of analyzers in the task they were designed for [23, 12, 22, 21, 3]. This paper offers a preliminary analysis of effectiveness of an analyzer which contributes to fill this gap and is novel in both the domain and the application: results are provided for an abstract interpreter based on the *sharing + freeness* domain presented in [17] and [7] in the application of automatic program parallelization.

The analyzer under study was designed to accurately and concisely infer at compile-time variable *groundness*, *sharing*, and *freeness* information for a program and a given query form. The abstract domain approximates this information by combining two components: one provides information on sharing (aliasing, independence) and groundness; the other encodes freeness information. Briefly, the former is essentially the abstract domain of Jacobs and Langen [11] (for efficiency and increased precision, however, the analyzer under study uses the efficient abstract unification and top-down driven abstract interpretation algorithms defined by Muthukumar and Hermenegildo [18] instead of the *pure bottom-up* approach used by Jacobs and Langen). The latter is represented as a list of those program variables which are known to be free.

Variable sharing is not only required in many types of analysis to ensure correctness, but is also quite useful in a number of applications and, in particular, essential in the compile-time detection of *strict independence* among goals (see [10] and its references), a condition which allows efficient parallelization of programs within the independent and-parallelism model. Informally, this condition states that a set of goals can run in parallel if they do not share any variable at run-time. Freeness information itself is also useful in a number of applications and essential in the detection of *non-strict independence* [10] among goals, a condition which extends strict independence. Furthermore, more accurate information is achieved in each of the domains by allowing communication between the two domains at some points of the analysis.

Both the accuracy of the information gathered by the interpreter and its effectiveness are evaluated during its use in the actual task of automatic parallelization of logic programs and while the interpreter is embedded in a real parallel logic programming system: &-Prolog [9]. These parameters are evaluated in terms of ultimate performance, i.e. the speedup obtained with respect to the sequential version of the program.

2. Overview of the Evaluation Environment

The *&-Prolog system* comprises a parallelizing compiler aimed at uncovering independent and-parallelism and an execution model/run-time system aimed at exploiting such parallelism. Prolog code is parallelized automatically by the compiler. Compiler switches determine whether or not code will be parallelized and through which type of analysis.

The *&-Prolog language* is a vehicle for expressing and implementing strict and non-strict independent and-parallelism. &-Prolog is essentially Prolog, with the addition of the parallel conjunction operator "`&`", a set of parallelism-related builtins, which includes several types of groundness and independence checks, and synchronization primitives. For syntactic convenience, an additional construct is also provided: the *Conditional Graph Expression (CGE)*. A CGE has the general form

$(i\_cond$ `=>` $goal_1$ `&` $goal_2$ `&` ... `&` $goal_N)$

where the $goal_i$ are either normal Prolog goals or other CGEs and i_cond is a condition which, if satisfied, guarantees the mutual independence of the $goal_i$s. The operational meaning of the CGE is "check i_cond; if it succeeds, execute the $goal_i$ in parallel, otherwise execute them sequentially."

There are three different annotators in the &-Prolog system: the CDG, the UDG and the MEL annotator, whose algorithms are defined in [16]. The CDG annotator seeks to maximize the amount of parallelism available in a clause, without being concerned with the size of the resultant &-Prolog expression. In doing this, the annotator may switch the positions of independent goals. *The UDG annotator* does essentially the same as the CDG annotator except that only unconditional parallelism is exploited, i.e., only goals which can be determined to be independent at compile-time are run in parallel. *The MEL annotator* seeks to maximize the number of goals to be run in parallel within a CGE, preserving the left-to-right order of subgoals in its expressions.

The two abstract interpreters which will be used in the evaluation are the *sharing + freeness* interpreter object of this study and the *sharing* only interpreter of [18].

The &-Prolog system can optionally generate a *trace file* during an execution. This file is an encoded description of the events that occurred during the execution of a parallelized program. Since &-Prolog generates all possible parallel tasks during execution of a parallel program, even if there are only a few processors in the system, all possible parallel program graphs, with their exact execution times, can be be constructed from this data. A tool has been devised and implemented which takes as input a real execution trace file of a parallel program run on the &-Prolog system, and gives as a result a new optimized trace file which corresponds to the *best possible* execution which would have occurred assuming a system with an infinite number of processors. It also provides statistics about the speedup obtained and the number of processors needed to achieve it. Since this "ideal" parallel execution uses as data a *real* trace execution file in which real execution times of sequential segments and all delay times are taken into account, it is possible to consider the results as a very good approximation to the *best possible* parallel execution.

Two broad categories of programs were used for the tests: simple programs and larger ones. Program selection within both categories has been performed taking into account the programs used in those studies with which the results of our tests are going to be compared.

3. Static Tests

We have first compared *statically* (i.e. lexically) the degree of parallelism and overhead introduced in a program which has been parallelized using the information of each analyzer and also using no information. We have parallelized several programs (see Table 1) using the MEL annotator.

Table 2 shows the results obtained when parallelizing programs with: no information (columns labeled with **w/o**), the information provided by the *sharing* analyzer (columns labeled with **sh**) and the information provided by the *sharing + freeness* analyzer (columns labeled with **sh+f**). The three main columns have respectively the following meaning: number of CGEs, number

Benchmark Programs	
browse	Gabriel benchmarks.
deriv	Symbolic differentiation. The expression given is: E+E-E*E/E*E/E where E is the addition of eight subexpressions
hanoi	Towers of Hanoi. The number of discs given is 9
prgeom	Constructs a perfect difference set of order n in increasing order, starting at 0.
qsrt	Quick sort algorithm with difference lists. Two lists have been given as input with lengths: 20 and 100
queens	It solves the n-queens problem.
serialize	Takes a list and converts each item to a number which is the position of that item in the sorted list. The list given as input has 25 characters
vmatrix	It multiplies an N by N matrix and an N by 1 matrix. The input is N = 10
Larger Programs	
asm	The SB-Prolog assembler
boyer	The theorem prover kernel in Gabriel Bench.
peephole	the peephole optimizer used in SB-Prolog
read	The public-domain Prolog tokenizer and parser.

Table 1. Programs used in the evaluation

of checks and number of unconditional CGEs in the whole parallelized program. The three subcolumns in the last two main columns also show in parenthesis the ratio between the number of checks and the total number of CGEs and the number of unconditional CGEs and the total number of CGEs, respectively.

	N. of CGEs			N. of checks			N. of uncond. CGEs		
program	w/o	sh	sh+f	w/o	sh	sh+f	w/o	sh	sh+f
asm	8	8	8	26 (3.25)	20 (2.5)	12 (1.5)	0 (0.0)	1 (12.5)	3 (37.5)
boyer	3	3	2	9 (3.00)	6 (2.00)	1 (0.50)	0 (0.0)	0 (0.0)	1 (50.0)
browse	5	5	5	12 (2.40)	10 (2.00)	10 (2.00)	0 (0.0)	1 (20.0)	1 (20.0)
peephole	3	3	2	13 (4.33)	9 (3.00)	3 (1.50)	0 (0.0)	1 (33.3)	1 (50.0)
prgeom	2	2	2	4 (2.00)	1 (0.50)	1 (0.50)	0 (0.0)	1 (50.0)	1 (50.0)
queens	3	3	2	9 (3.00)	2 (0.66)	0 (0.00)	0 (0.0)	1 (33.3)	2 (100.0)
serialize	1	1	1	4 (4.00)	4 (4.00)	1 (1.00)	0 (0.0)	0 (0.0)	0 (0.0)
read	5	5	1	15 (3.00)	5 (1.00)	1 (1.00)	0 (0.0)	0 (0.0)	0 (0.0)
vmatrix	3	3	3	10 (3.33)	1 (0.33)	0 (0.00)	0 (0.0)	2 (66.6)	3 (100.0)
deriv	4	4	4	20 (5.00)	4 (1.00)	0 (0.00)	0 (0.0)	0 (0.0)	4 (100.0)
hanoi	1	1	1	4 (4.00)	0 (0.00)	0 (0.00)	0 (0.0)	1 (100.0)	1 (100.0)
qsrt	1	1	1	1 (1.00)	0 (0.00)	0 (0.00)	0 (0.0)	1 (100.0)	1 (100.0)

Table 2. Static Results for the Sharing and Sharing + Freeness Analyzers

3.1. Dynamic Tests

An arguably better way of measuring the effectiveness of the information provided by abstract interpretation-based analyzers is to measure the speedup achieved in the parallel execution time of the program (ideally for an unbounded number of processors) against the sequential program execution time, while using the information provided by such analyzers in the parallelization. This ideal parallel execution time has been obtained using the tools described in section 2.

A related type of test has also been described in [20]. This paper presents a high-level simulation study of the amount and characteristics of the or- and (independent) and-parallelism in a wide selection of Prolog programs. In that study, simple programs were parallelized by hand and the others were parallelized with the MEL annotator first, and then optimized by hand. The results presented there will be used here as a reference for the maximum parallelization that can be achieved.

The results are presented in table 3. This table shows the speedup obtained by the parallelized program w.r.t. the sequential execution of the sequential program and the number of processors needed to obtain it (presented after the @ symbol). The results were obtained with checks implemented in C. The second block shows the results obtained when parallelizing the programs by hand (labeled as **hand p.**), and the corresponding results presented in [20] (labeled as **s. K + H**). The first block of the table is divided into three main rows labeled as **MEL**, **CDG** and **UDG**, which indicate the annotator used for each test. Each main row is also divided in three rows whose labels show the type of analysis used for the parallelization.

		deriv	**hanoi**	**vmatrix**	**qsrt(20)**	**qsrt(100)**	**serialize**
MEL.	**w/o**	0.82 @ 208	17.80 @ 282	1.00 @ 20	0.8 @ 7	1.78 @ 25	0.88 @ 4
	sh	23.54 @ 237	41.77 @ 466	3.75 @ 24	1.58 @ 13	2.79 @ 70	0.88 @ 4
	sh+f	42.49 @ 248	41.77 @ 466	5.80 @ 28	1.58 @ 13	2.79 @ 70	0.97 @ 4
CDG.	**w/o**	0.83 @ 207	21.30 @ 271	1.21 @ 20	1.14 @ 7	1.92 @ 23	0.96 @ 5
	sh	28.03 @ 239	41.77 @ 490	4.19 @ 22	1.57 @ 8	2.79 @ 22	0.96 @ 5
	sh+f	42.49 @ 256	41.77 @ 490	6.24 @ 23	1.57 @ 8	2.79 @ 22	1.08 @ 4
UDG.	**w/o**	1.0 @ 1	1.0 @ 1	1.0 @ 1	1.0 @ 1	1.0 @ 1	1.0 @ 1
	sh	1.0 @ 1	41.77 @ 490	1.04 @ 12	1.57 @ 8	2.79 @ 22	1.0 @ 1
	sh+f	42.49 @ 256	41.77 @ 490	6.24 @ 23	1.57 @ 8	2.79 @ 22	1.0 @ 1
hand p.		42.49 @ 256	41.77 @ 490	6.24 @ 23	1.57 @ 8	2.79 @ 22	1.09 @ 4
s. K + H		84.5 @ 248	52.3 @ 427	9.06 @ 18	1.56 @ 3	2.80 @ 8	1.08 @ 4

Table 3. Speed up w.r.t. the Sequential Execution of the Sequential Program

3.2. Discussion of the Results

Looking at the static results presented in table 2, the first point that can be observed is that the number of resulting CGEs can decrease (`boyer`, `peephole`, `queens` and `read`) if the information provided by the *sharing + freeness* analyzer is considered. This is due to the freeness information: ground checks over the variable X can be known to fail if X is known to be free at this point of the execution, therefore eliminating the CGE and executing the goals sequentially without tests.

The second point is the improved accuracy of the information provided by the *sharing + freeness* analyzer: the results obtained with this analyzer are always better than those obtained without analysis, and equal or better than those obtained with the *sharing* analyzer. This confirms that communication between abstract domains during the analysis increases the accuracy of the resulting information.

It can be thought that although the results of table 2 show that sometimes the *sharing + freeness* analyzer is significatively better than the *sharing* analyzer (e.g. the results obtained for *asm* and *peephole*), in the rest only a few checks are eliminated (four in the best case). However, it turns out that eliminating only one check may produce a great difference in the speedup achieved: the dynamic tests show for *vmatrix* a speedup of up to a factor of 6 with only one check of difference.

Before discussing the results obtained in the dynamic tests, a few points should be made. Firstly, some of the programs used in this test had to be kept small in size. This is due to the fact that they have small granularity and generate a very large number of tasks (in the order of 10^5) and reach the hardware and software limitations of our "ideal speedup" tools. Secondly, since the dynamic tests have been performed with *real* executions, they always include some number of interruptions (due to the Unix Operating System over which the tools are executing) in the parallel execution which do not allow achieving a real *maximum* parallel execution time. Furthermore, these interruptions produce significant variations among the execution times obtained for the same program. Therefore, the results, which have been taken as the minimum of the times obtained in different (10) executions, can lead to somewhat surprising results when execution time is short as for example in `qsrt(20)`: while the speedup obtained parallelizing by hand is 1.57, automatic parallelization can achieve 1.58. This is simply due to "noise" in the measurements over real systems.

The results of dynamic tests show the importance of the information provided by the *sharing + freeness* analyzer and its accuracy, since its results are always equal or better than the rest, and sometimes much better. This is particularly evident for the *vmatrix* and *deriv* programs, in which speedup is significatively higher. As mentioned before, the first case is quite interesting since the difference with the information provided by the *sharing* analyzer results in the elimination of only one check.

It is important to note that in most of the cases (all but *serialize*) hand parallelization obtains the same results as the analyzer. It is somewhat surprising that the speedup of the hand parallelization achieved with the tools used herein is somewhat different in the first three programs from the speedup obtained with the simulator described in [20]. The answer can be in the fact that the simulator of [20] takes as time reference the number of head unifications, considering all head unifications of equal cost, rather than actual execution times. Also, the costs of all builtin predicates appearing in the program are considered equal. We can see in the tables that the differences between the speedups are larger in those programs in which the amount of and-parallelism is high and has a balanced and not small granularity (thus maintaining the processors active most of the time). It is also interesting to note that actual speedup is achieved with parallelizations in which not all checks have been eliminated. This, coupled with the increased accuracy of the new analyzers, makes the UDG algorithm perhaps not as preferable as it had seemed at first glance. The results also confirm the superiority of the CDG annotator comparing to the other two. However, the results are better at the cost of a great number of processors.

References

[1] M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. Technical Report CW62, Department of Computer Science, Katholieke Universiteit Leuven, October 1987.

[2] M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inference. In *5th Int. Conf. and Symp. on Logic Prog.*, pages 669–683. MIT Press, August 1988.

[3] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. In *Fourth IEEE International Conference on Computer Languages (ICCL'92)*, San Francisco, CA, April 1992.

[4] C. Codognet, P. Codognet, and M. Corsini. Abstract Interpretation of Concurrent Logic Languages. In *North American Conference on Logic Programming*, pages 215–232, October 1990.

[5] M. Corsini and G. Filè. The abstract interpretation of logic programs: A general algorithm and its correctness. Research report, Department of Pure and Applied Mathematics, University of Padova, Italy, December 1988.

[6] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conf. Rec. 4th Acm Symp. on Prin. of Programming Languages*, pages 238–252, 1977.

[7] M. Garcia de la Banda and M. Hermenegildo. Effectiveness of combined sharing and freeness analysis using abstract interpretation. Technical report, U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, January 1992.

[8] S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. *Journal of Logic Programming*, pages 207–229, September 1988.

[9] M. Hermenegildo and K. Greene. The &-prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[10] M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

[11] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

[12] A. Marien, G. Janssens, A. Mulkers, and M. Bruynooghe. The Impact of Abstract Interpretation: an Experiment in Code Generation. In *Sixth International Conference on Logic Programming*, pages 33–47. MIT Press, June 1989.

[13] K. Marriott and H. Søndergaard. Semantics-based dataflow analysis of logic programs. *Information Processing*, pages 601–606, April 1989.

[14] C.S. Mellish. Abstract Interpretation of Prolog Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 463–475. Imperial College, Springer-Verlag, July 1986.

[15] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

[16] K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *1990 International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[17] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*. MIT Press, June 1991.

[18] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.

[19] T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34:227–240, 1984.

[20] K. Shen and M. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In *1991 International Logic Programming Symposium*. MIT Press, October 1991.

[21] A. Taylor. LIPS on a MIPS: Results from a prolog compiler for a RISC. Technical report, Association for Logic Programming, June 1990.

[22] P. Van Roy and A. M. Despain. The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler. In *Proceedings of the North American Conference on Logic Programming*, pages 501–515. MIT Press, October 1990.

[23] R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*. MIT Press, August 1988.