GRANLOG: An Integrated Granularity Analysis Model for Parallel Logic Programming

Jorge Luis Victória Barbosa [a,b], Patrícia Kayser Vargas [b], Cláudio Fernando Resin Geyer [b] and Inês de Castro Dutra [c]

[a] *Universidade Católica de Pelotas, Escola de Informática*
Pelotas, RS, Brazil
barbosa@atlas.ucpel.tche.br

[b] *Universidade Federal do Rio Grande do Sul, Instituto de Informática*
Porto Alegre, RS, Brazil
{barbosa,kayser,geyer}@inf.ufrgs.br

[c] *Universidade Federal do Rio de Janeiro, COPPE/Sistemas*
Rio de Janeiro, RJ, Brazil, ines@cos.ufrj.br

Abstract

This work presents a new granularity analysis model for logic programming, the GRANLOG model. GRANLOG can generate granularity information for different sources of parallelism, and this information can be used by different parallel logic programming systems. GRANLOG was designed and implemented in modules that facilitate the inclusion and modification of any part of the system. GRANLOG employs several static analysis techniques based on other global analysis systems such as MEL and UDG and extends other tools used for granularity analysis such as CASLOG. We show that GRANLOG can effectively help logic programming systems to exploit parallelism in a more efficient way exhibiting some preliminary results obtained with the utilization of GRANLOG.

Key words: Logic programming, parallelism, granularity analysis

---

1 Introduction

The exploitation of implicit parallelism, that is relatively easier to detect in logic-based languages than in imperative languages, is considered an alternative that simplifies the parallel machines programming and which improves the logic programs performance. Logic programming allows the exploitation of two common forms of implicit parallelism: and-parallelism, where several goals in a conjunction can be selected to execute in parallel; and or-parallelism where several candidate clauses for a goal can be evaluated in parallel.

Several systems exploit only or-parallelism such as PloSys [8], Muse [14], Aurora [15] and YapOR [16]. Other systems exploit only and-parallelism such as &-Prolog [17], DASWAM [18], &-ACE [27] and Reform Prolog [24]. Yet other systems can exploit both kinds of parallelism such as Andorra-I [6,7], Penny [19] or ParAKL [20]. Such systems generate quite irregular parallel execution patterns that demand dynamic load balancing in order to maintain all processors busy most of the execution time. Most schedulers in these systems which distribute or-work, and-work or and-or work [25] make scheduling decisions based on runtime information and use different heuristics to estimate the amount of parallel work available in the execution tree. Parallel execution can be improved by a combination of runtime decisions and information generated at compile-time since the scheduler could have more precise information about amount of parallelism to help it to distribute work.

Most work developed in the area of compile-time analysis has concentrated on determinacy, mode and argument type analysis for generating optimized code mainly to sequential logic programming systems. In this work we develop a framework for compile-time granularity analysis in order to generate information for parallel logic programming systems that exploit and-parallelism, or-parallelism and and-or-parallelism. Compile-time granularity analysis consists of evaluating sizes of goals and clauses and amount of potential parallelism. Research has been done in compile-time granularity analysis for concurrent logic languages such as FGHC [21] and for parallel logic programming systems that exploit independent [22] and dependent [28] and-parallelism. Besides, Debray et al have developed the CASLOG system to perform complexity analysis of goals and clauses [4][5].

In this work we present a more complete framework for compile-time granularity analysis and its implementation, GRANLOG (GRanularity ANalyzer for LOGic programs).

GRANLOG performs automatic granularity analysis of Prolog programs and generates annotated programs. It uses several advanced static analysis techniques such as Global Analysis via Abstract Interpretation [23][1][2] and Complexity Analysis [3][4][5]. The information generated by the system can be used in applications such as scheduling decisions and program execution simulation. In order to demonstrate the usefulness of GRANLOG, scheduling strategies based on information generated by the system were implemented. These strategies were applied in two systems that exploit parallelism in logic programs: Andorra-I [6][7] and PloSys [8]. They were also used in the DSLP simulator [9] which is a prototype used to evaluate the DSLP distributed scheduler.

The remainder of the paper is organized as follows. In section 2, the basic principles of the system are presented. Section 3 shows an overview of GRANLOG focussing on its modular structure. Sections 4, 5 and 6 describe the modules that compose the system: the Global Analyzer, the Grain Analyzer and the Complexity Analyzer. In section 7, we present the improvements that GRANLOG can produce on Andorra-I, PloSys and DSLP. Finally, section 8 draws conclusions and discusses future work.

2 Design Decision

We designed and implemented the GRANLOG system having in mind the following principles:

Principle 1: *Independence of the parallel system that will use the information.* The granularity information, which is obtained through the analysis performed by GRANLOG does not depend on the parallel system (parallel architecture and runtime environment) on which the program will be executed. In this way, flexibility and portability are achieved.

Principle 2: *Automatic detection of parallelism (implicit parallelism).* The system implements the automatic detection of parallelism, concerning logic programs. Therefore, end-programmers do not get involved with parallelism exploitation and the programming language keeps unchanged. This principle allows reuse of already developed programs and frees end-programmers from managing parallel development tasks.

Principle 3: *Optimal exploitation of parallelism in logic programs.* GRANLOG generates information on granularity analysis related to or-parallelism and and-parallelism. GRANLOG tries to obtain the maximum amount of parallelism information arising from the most common sources of parallelism found in logic programs to allow an optimum exploitation of parallelism.

3 GRANLOG Structure

GRANLOG is composed by three modules: the Global Analyzer (GLA), the Grain Analyzer (GRA) and the Complexity Analyzer (CA). Figure 1 presents the basic organization of the system. The input is a Prolog program and the output is a granulated program. The *granulated program* is composed by the original program plus *granularity annotation*. This annotation expresses the information produced by the granularity analysis process.

GLA employs global analysis to determine modes, types and argument sizes (named measures in the figure) for each predicate. Modes establish the direction of the argument bindings (*input*, *output*, *input/output*). Argument sizes are used during the complexity analysis to determine the influence of each argument in the complexity of the predicate. Types determine the binding pattern of the arguments (*list*, *integer*, *atom*, etc). Moreover, GLA analyzes accurately the dependencies between clause literals. Based on this analysis, the Grain Analyzer module establishes the grains of a clause body, i.e., which are the tasks that can be executed in parallel. The information inferred by GRA is annotated in the original program. The resulting program receives the name of *partitioned program*.

The Complexity Analyzer evaluates the cost involved in the execution of the grains (granularity) detected by the Grain Analyzer. In addition, it determines the complexity of the *pending resolvent* for each literal in a clause. The pending resolvent consists of all continuation literals that follow a clause literal [3]. It is used to get the complexity of a search tree branch during the execution of programs (*OR complexity* [3]). The CA module also determines the relationship between the input and output sizes of each grain (evaluation of communication costs). The information inferred by CA is annotated in the partitioned program. The final product is the granulated program. In the following sections we describe each module in more detail.

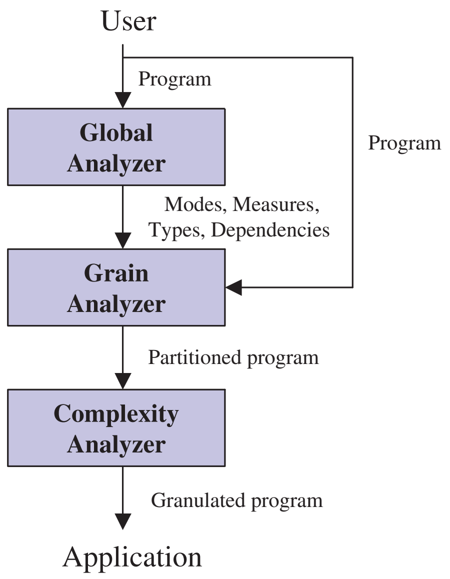

Fig. 1. GRANLOG structure

4 The Global Analyzer Module

The Global Analyser (GLA) is the first module in the GRANLOG system. It was designed in two parts that were implemented as two independent modules: *Party* (PARrallel TYpes analyzer [10]) and *MODA* (MOdes and Dependencies Analyzer [1]). ParTy is an abstract interpreter that uses *abstract compilation* [2] to determine the types of arguments in a Prolog program. On the other hand, MODA is an abstract interpreter of modes and dependencies. Party and MODA are independent systems that were integrated to create GLA [1]. Figure 2 presents the internal structure of the Global Analyzer. GLA receives a Prolog program increased with an *entry query* (or entry point) that describes basic characteristics of the predicate used to call the program. An entry query indicates the most used query pattern for a program with information that makes global analysis possible. Detailed description of the entry query will not be discussed in this paper due to space restrictions. After the analysis, the module produces information about the abstract execution of the program.

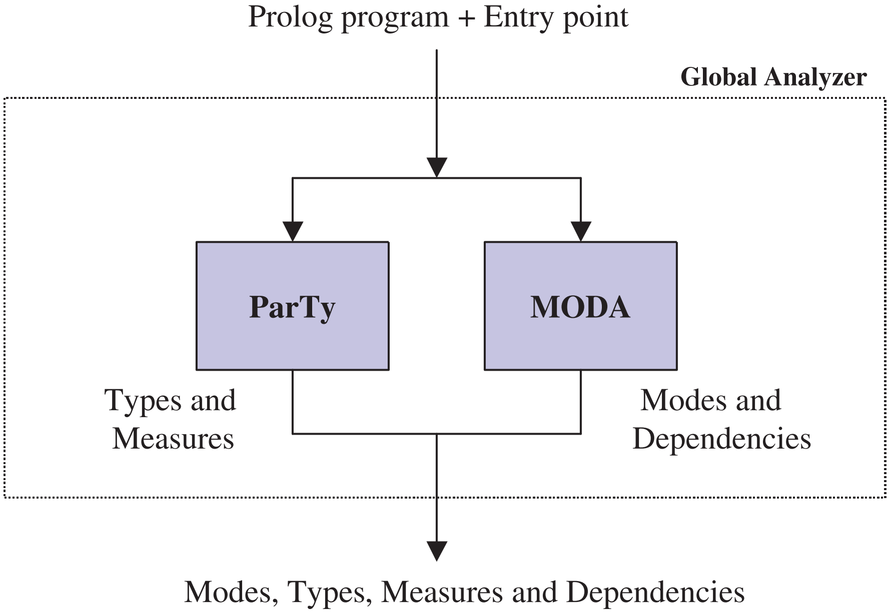

Fig. 2. Global Analyzer structure

Figure 3 shows the program *Hanoi* that will be used to explain the results of Global Analysis. This example program is composed by two predicates: `hanoi/5` and `append/3`. As a result of global analysis of this program, information related to each of the clauses of both predicates is generated. As an example, figure 4 presents the output pattern generated by MODA to the second clause of the predicate `hanoi/5`. In the figure, modes and dependencies are organized in two groups. The first group (first four lines) contains information about modes of each body literal (information is generated only for user predicates). Each line identifies the literal and presents a list with modes for all arguments (i indicates *input* and o indicates *output*).

The representation is `predicatename/arity/clausenumber/goalnumber`. The second group (last line) describes the dependencies among the clause literals. The information is structured as an annotation that identifies the clause and presents a list with pairs of literal dependencies (I and O are used to indicate the head).

This notation was designed without using any specific system as a model. The great contribution of this module is to use several important and consolidated concepts in a sole new output notation. For example, the notion of output and input modes in Prolog has been studied for a long time [2,4,5] and the dependency graph was used in early models as Conery's AND/OR Process Model [26].

Figure 5 presents the graph dependency created by the global analysis for this example.

```
hanoi(1,A,B,C,[mv(A,C)]).
hanoi(N,A,B,C,M) :-
   N > 1, N1 is N - 1,
   hanoi(N1,A,C,B,M1), hanoi(N1,B,A,C,M2),
   append(M1,[mv(A,C)],T), append(T,M2,M).

append([],L,L).
append([H|L],L1,[H|R]) :- append(L,L1,R).
```

Fig. 3. Program Hanoi

```
hanoi/5/2/3 [i,i,i,i,o]
hanoi/5/2/4 [i,i,i,i,o]
hanoi/5/2/5 [i,i,o]
hanoi/5/2/6 [i,i,o]
:-depend(hanoi/5/2,[(I,1),(I,2),(2,3),(2,4),(3,5),(4,6),(5,6),(6,O)]).
```

Fig. 4. Output generated by MODA

Following the example, figure 6 shows a possible output generated by ParTy for the predicate `hanoi/5`. The first line presents the type of each argument that was inferred considering one specific entry query. The first argument of the hanoi predicate was found to be of integer type. The second, third and fourth arguments are all atoms whose size is found to be three characters. The last argument is a list of unknown size (in the figure, represented by "?") but with known element type: structure whose name has two characters (mv in the program) with arity equals 2 and both sub-arguments of type *atom(3)*. When all sub-arguments have the same type, this is presented only once, otherwise, a complete list of types is included in the type annotation. The second line keeps the measures that must be used during the complexity analysis. The first argument must be measured like an integer (*int*) and the last must be measured considering the number of list elements (*length*). The first integer controls the hanoi recursion while the list size determines the output size. The other arguments are not used by the complexity analysis (*void*) since they do not influence the complexity of this program .

The combination of modes and types defines *communication channels*. Each argument defines a channel that can be used to predict the flow of

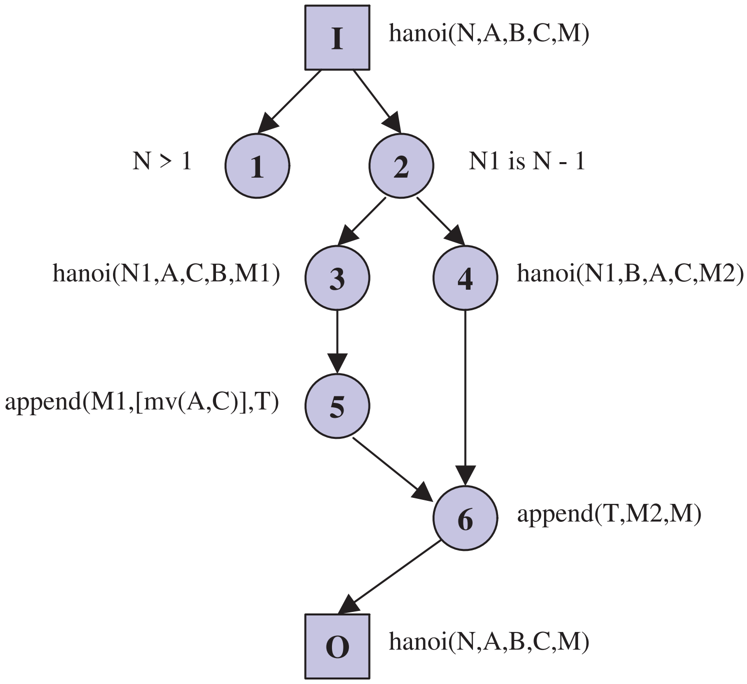

Fig. 5. Graph dependency generated by MODA

```
:- type(hanoi/5,[int,atom(3),atom(3),atom(3),list(?,[struct(2,2,
   [atom(3)])])]).
:- measure(hanoi/5,[int,void,void,void,length]).
```

Fig. 6. Output generated by Party

information exchanged by processors during the parallel execution. Using this prediction, it is possible to calculate the size of messages (communication costs) in a distributed memory system. For example, figure 7 shows a possible configuration of channels during the parallel execution of hanoi. Modes define the directions and types determine the kind of information in the message. In this example, there are four input (mode i) and one output channels (mode o). Input messages have well defined types. Therefore, sizes and costs can be precisely evaluated. On the other hand, the information about the output message is incomplete, because the list size is unknown. The list size depends on the first argument which value will be sent in the first input channel during the execution. It is important to consider these channels as logical channels. None efficient implementation would use five different messages for sending and receiving arguments.

5 Grain Analyzer

The Grain Analyzer (GRA) module uses the dependencies information generated by the Global Analyzer to create the *partitioned program*. GRA inserts in

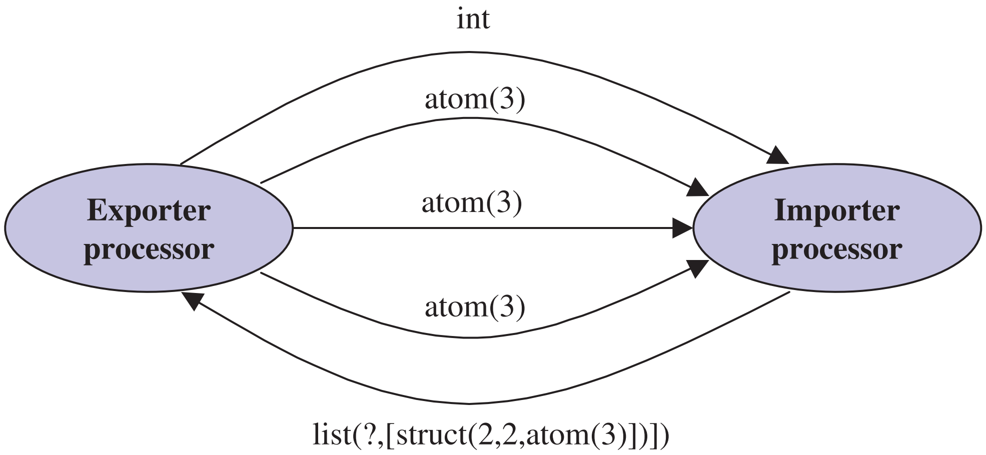

Fig. 7. Communication channels using modes and types

the original program an annotation that expresses the potential parallelism. This annotation identifies the grains and describes their characteristics. In the GRANLOG context, a grain is defined as a task that can be executed in parallel with other tasks.

Figure 8 shows the same program example, hanoi, after annotated by the GRA module. The symbol & is used to delimit the grains in the body of each clause. Before the original source program, GRA introduces *grain annotations* that describe the characteristics of each grain. There are three kinds of grain annotation: *grain_clause*, *grain_goal* and *grain_goals*. The *grain_clause* annotation describes the characteristics of a grain related to a clause. The *grain_goal* annotation describes the characteristics of a grain related to a body clause goal. Finally, the *grain_goals* annotation describes the characteristics of a grain related to the goals of a body clause.

The parameters of *grain_clause* are the predicate name and arity, the grain identifier and the arguments modes and measures. The *grain_goal* annotation has an additional parameter, which contains arguments types. The *grain_goals* annotation has all parameters used by *grain_goal* plus one extra parameter that contains the argument names (input and output grains). The Grain Analyzer evaluates the grains associated to a set of goals in the body clause and identifies their inputs and outputs.

The Grain Analyzer indicates the potential parallelism available from clauses of a predicate (OR parallelism) and from each clause body (AND parallelism). OR parallelism is exploited using the information provided by the *grain_clause* annotation. AND parallelism is exploited using the symbol & and the annotations *grain_goal* and *grain_goals*. Further, GRA reorganizes goals in the body clause in order to exploit as much parallelism as possible. Figure 8 exemplifies this situation where the first call to the `append/3` goal in the second hanoi

```
:- grain_clause(hanoi/5,g1,[i,i,i,i,o],[int,void,void,void,length]).
:- grain_clause(hanoi/5,g2,[i,i,i,i,o],[int,void,void,void,length]).
:- grain_goals(hanoi/5,g2_3,[N1,A,B,C,T],[i,i,i,i,o],[int,void,void,
   void,length],[int,atom(3),atom(3),atom(3),list(?,[struct(2,2,
   [atom(3)])])]).
:- grain_goal(hanoi/5,g2_3_1,[i,i,i,i,o],[int,void,void,void,length],
   [int,atom(3),atom(3),atom(3),list(?,[struct(2,2,[atom(3)])])]).
:- grain_goal(hanoi/5,g2_3_2,[i,i,o],[length,void,length],
   [list(?,[struct(2,2,[atom(3)])]),list(1,[struct(2,2,[atom(3)])]),
   list(?,[struct(2,2,[atom(3)])])]).
:- grain_goal(hanoi/5,g2_4,[i,i,i,i,o],[int,void,void,void,length],
   [int,atom(3),atom(3),atom(3),list(?,[struct(2,2,[atom(3)])])]).
:- grain_goal(hanoi/5,g2_5,[i,i,o],[length],[list(?,[struct(2,2,
   [atom(3)])])]).
hanoi(1,A,B,C,[mv(A,C)]).
hanoi(N,A,B,C,M) :-
   N > 1, N1 is N - 1,
   (hanoi(N1,A,C,B,M1),append(M1,[mv(A,C)],T)) &
   hanoi(N1,B,A,C,M2), append(T,M2,M).
```

Fig. 8. Predicate hanoi after grain analysis

clause of the original source program was replaced with the second call to `hanoi/5`.

6 Complexity Analyzer

The Complexity Analyzer module evaluates the computational costs involved in the execution of the grains identified by the Grain Analyzer. CA is composed by two modules: *ORCA* (OR Complexity Analyzer [3]) and *CASLOG* (Complexity Analysis System for LOGic [4][5]). Both modules generate information about the complexity of logic programs.

ORCA produces simplified information used to determine the OR complexity. It is based on Tick's algorithm [11] with some modifications to increase precision as presented in [3]. CASLOG was developed by Lin and Debray and generates highly accurate and complex recurrent expressions that can be used to determine the complexity of each predicate in a program (AND complexity). The information generated by ORCA is called simplified because it does not consider actual recursion and gets all information at compile time. In contrast, CASLOG generates expressions which can be evaluated at run time and give really precise complexity information.

Figure 9 shows the internal structure of the Complexity Analyzer module.

The Complexity Analyzer includes *complexity annotations* in the partitioned program. Figure 10 presents the modifications performed in the partitioned hanoi program (figure 8). The *granularity* annotation contains the granularity expression. This expression can be used to calculate the computa-

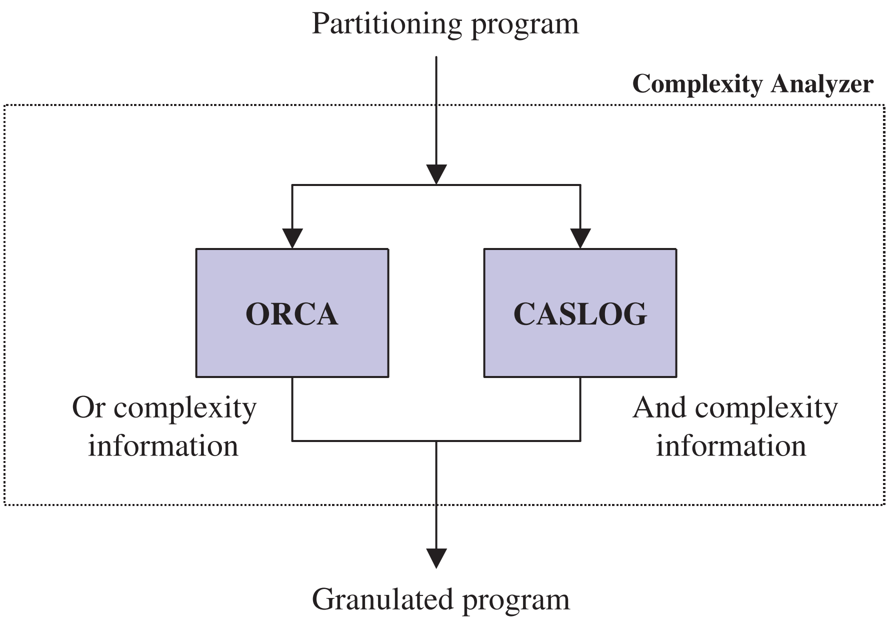

Fig. 9. Complexity Analyzer structure

tional cost of a grain considering the size of its inputs. The two parameters of *granularity* are an expression identifier (*e1*, for example) and the expression itself. The *out_size* annotation contains the relation between the sizes of the input and output grains. This relation can be used to determine the output size of a grain regarding its input size. In this way, it is possible to compute the cost for achieving the grain results in distributed systems (message size to get output). For example, in figure 7 this was the unique information that was not available at compile-time.

The two parameters of *out_size* consist of a relation identifier (*r1*, for example) and the relation itself. The notation used in the expressions is the same as those utilized in CASLOG [4][5]. In this notation, we use mathematical operators (*,/,+,-), mathematical functions (*fact*, *exp* and *log*, for example) and the *$* symbol. The latter indicates the size of a grain argument (*$1*, for example, represents the size of the first argument of the grain). Considering the *out_size* annotation, the relation between the argument sizes of a grain is described by the notation of the second parameter. For example, the notation *[$1,$2,$1+$2]* means that the grain has three arguments. The size of the third argument (output) is calculated using the first (*$1*) and the second (*$2*) argument sizes through the expression *$1+$2*.

The Complexity Analyzer introduces three new parameters in the grain annotations. The first one (for example, 39 in the second *grain_clause* annotation) contains a constant produced by ORCA. This constant is used to perform the OR complexity analysis. In the *grain_clause* annotation, this num-

```
:- granularity(e1,$1*exp(2,$1)+exp(2,$1-1)-2).
:- granularity(e2,$1*exp(2,$1)+3*exp(2,$1-1)-2).
:- granularity(e3,$1+1).
:- out_size(r1,[0,0,0,0,1]).
:- out_size(r2,[$1,0,0,0,exp(2,$1)-1]).
:- out_size(r3,[$1,0,0,0,exp(2,$1)]).
:- out_size(r4,[$1,0,$1+1]).
:- out_size(r5,[$1,$2,$1+$2]).
:- out_size(r6,[0,$2,$2]).
:- grain_clause(hanoi/5,g1,[i,i,i,i,o],[int,void,void,void,length],
   1,1,r1).
:- grain_clause(hanoi/5,g2,[i,i,i,i,o],[int,void,void,void,length],
   39,e1,r2).
:- grain_goals(hanoi/5,g2_3,[N1,A,B,C,T],[i,i,i,i,o],[int,void,void,
   void,length],[int,atom(3),atom(3),atom(3),list(?,[struct(2,2,
   [atom(3)])],19,e2,r3).
:- grain_goal(hanoi/5,g2_3_1,[i,i,i,i,o],[int,void,void,void,length],
   [int,atom(3),atom(3),atom(3),list(?,[struct(2,2,[atom(3)])])],24,e1,r2).
:- grain_goal(hanoi/5,g2_3_2,[i,i,o],[length,void,length]),
   [list(?,[struct(2,2,[atom(3)])]),list(1,[struct(2,2,[atom(3)])]),
   list(?,[struct(2,2,[atom(3)])])],19,e3,r4).
:- grain_goal(hanoi/5,g2_4,[i,i,i,i,o],[int,void,void,void,length],
   [int,atom(3),atom(3),atom(3),list(?,[struct(2,2,[atom(3)])])],5,e1,r2).
:- grain_goal(hanoi/5,g2_5,[i,i,o],[length],
   [list(?,[struct(2,2,[atom(3)])])],0,e3,r5).
```

Fig. 10. Complexity annotations inserted by the Complexity Analyzer

ber represents the complexity of a clause. In the *grain_goal* and *grain_goals* annotations, the number represents the complexity of all next goals in the clause (pending resolvent). The second parameter (for example, *e1* in the second grain_clause annotation) consists of an identifier. It points to the expression that can be used to calculate the computational cost of the grain. This parameter binds the grain annotations with the granularity annotation. The third new parameter (last) consists of the identifier of the relation, which determines the grain output considering its inputs (*r1*, for example). This parameter binds the grain annotations with the *out_size* annotation. Some *granularity* and *out_size* annotations are used by more than one grain. This fact is very meaningful because they employ the same binding (for example, *g2*, *g2_3_1* and *g2_4* grains, which use the same granularity expression). Sometimes, the Complexity Analyzer can infer that a grain has a computational cost, which is constant. In this case, the granularity annotation will not be used. On the other hand, the constant will be annotated in the penultimate parameter of grain annotations (for example, *g1* grain).

7 GRANLOG Applications

GRANLOG was used to improve the exploitation of parallelism in three systems, namely: Andorra-I [6][7], PloSys [8] and DSLP [9].

Andorra-I is a parallel logic programming system that exploits both determinate AND parallelism and OR parallelism in shared-memory parallel computers. In Andorra-I, the compiler was adapted to transfer the granularity information from granulated program to abstract code (VRAM code [7]). The system uses the simplified information generated by ORCA module. Moreover, the engine was modified to access and update the runtime granularity information. This mainly consisted of adapting the loader and supporting the new VRAM instruction fields.

Regarding the scheduling strategies, modifications were performed to provide information to the reconfigurer only. The Granlog-Andorra integration presented stimulating preliminary results. We used 6 applications containing and-parallelism, or-parallelism and and-or parallelism and ran two versions of Andorra-I on a Sun SPARCstation-20 with 4 processors. Our results show that the GRANLOG information helped improving performance on all but one application. Our improvement ranged from 1% to 168%. These results were published in another paper [12].

PloSys explores OR parallelism using parallel machines with distributed memory. We needed to modify the abstract Plosys instructions to support granularity control. The instructions used to control the OR nodes now compute the complexity of choice-points using complexity of goals and pending resolutions. In addition, new algorithms were created to improve scheduling decisions using this information. In its original version, the scheduler strategy used the *number of choice-points* to guide its decisions. Now it uses dynamic computed complexity [13] and the implementation presented its first results that need to be improved.

DSLP (Distributed Scheduling for Logic Programming) is a distributed scheduler for exploiting OR parallelism and independent AND parallelism in logic programming. It was designed to work on shared memory and distributed memory systems. It uses two main informations to aid scheduling: system information (detected before and during the execution) and granularity information generated by GRANLOG.

The system was originally designed to support granularity control in both kinds of parallelism (AND/OR). Currently, DSLP has only a simulation version to control AND parallel execution. It uses the same information manipulated by Andorra-I. All scheduler decisions are guided by the *complexity of goals*. The first results are encouraging and show that the model can use the granularity information to increase the performance [9].

8 Conclusions

This work presented an integrated model for automatic granularity analysis of logic programs. Its main contributions are the integration of several concepts of granularity analysis in a single framework, the modeling, modularisation and implementation of these concepts, and the introduction of new annotation structures to encapsulate important information needed to improve the efficiency of scheduling strategies. GRANLOG integrates compile-time granularity analysis information with run-time information in order to provide more qualified data to support scheduling decisions.

We have been using GRANLOG information in systems such as Andorra-I [12], PloSys [13] and DSLP [9] with good results and minimum programming effort.

We plan to further investigate the impact of the GRANLOG information in these systems, and in the near future we intend to use GRANLOG to support scheduling decisions in other parallel logic programming systems such as YAPOR [16] and ACE [27]. One advantage of using GRANLOG is that we can improve modules separately depending on the systems needs.

We also intend to extend GRANLOG to perform better source code reorganization and program simulation.

Acknowledgments

We would like to thank Ana Paula Centeno, Cristiano André da Costa, Débora Nice Ferrari, Edvar B. Araújo, Luís Fernando Pias de Castro, Silvana C. de Azevedo and Thobias Salazar for the discussions and their helpful contributions. This research was partially supported by Protem-CC/CNPq during APPELO Project research grant and by PPGC/UFRGS and COPPE-Sistemas/UFRJ.

References

[1] S. C. Azevedo and J. L. V. Barbosa and C. F. R. Geyer, Automatização da Análise Global no modelo Granlog, *XXV Conferência Latino Americana de Informática (CLEI)* (Asuncion, Paraguay, August-September,1999) 601-612.

[2] S. Debray and R. Ramakrishnan, Abstract Interpretation of Logic Programs Using Magic Transformations, *Journal of Logic Programming* **18** (1994) 149-176.

[3] J. L. V. Barbosa and C. F. R Geyer, Análise de Complexidade na Programação em Lógica: Taxonomia, Modelo Granlog e Análise OU, *XXIII Conferência Latino Americana de Informática (CLEI)* (November 1997) 609-618.

[4] S. K. Debray and N. Lin, Cost Analysis of Logic Programs, *ACM Transactions on Programming Languages and Systems* **15** (1993) 826-875.

[5] N. Lin, *Automatic Complexity Analysis of Logic Programs* (PhD Thesis, University of Arizona, Tucson, 1993).

[6] R. Yang and T. Beaumont and I. C. Dutra and V. Santos Costa and D. H. D. Warren, Performance of the Compiler-Based Andorra-I System, *Proceedings of the Tenth International Conference on Logic Programming* (MIT Press, June 1993) 150-166.

[7] V. Santos Costa, *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I* (PhD Thesis, University of Bristol, 1993).

[8] E. Morel and J. C. Kergommeaux and C. F. R. Geyer, Side-effects in PloSys OR-parallel Prolog on Distributed Memory Machines, *Post-Conference Workshop on Parallelism and Implementation Technology for (Constrain) Logic Programming Languages (JICSLP'96)* (Bonn, Germany, September 1996).

[9] C. A. Costa and C. F. R. Geyer, Uma proposta de escalonamento distribuído para exploração de paralelismo na programação em lógica, *X Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho (SBA C-PAD)* (SBC, Búzios, Brazil, 1998) 61-64.

[10] L. F. Castro and C. F. R. Geyer, ParTy – A Parallel Types Analyzer, *Workshop on Parallelism and Implementation Technology for Logic Programming Languages* (October 1997).

[11] E.Tick, Compile Time Granularity Analysis for Parallel Logic Programming Systems, *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems* (ICOT, 1988).

[12] I. C. Dutra and V. Santos Costa and J. L. V. Barbosa and C. F. R. Geyer, Using Compile-Time Granularity Information to Support Dynamic Work Distribution in Parallel Logic Programming Systems, *XI Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho (SBAC-PAD)* (SBC, Natal, Brazil, 1999) 248-254.

[13] D. N. Ferrari and P. K. Vargas and C. F. R. Geyer and J. L. V. Barbosa, Modelo de Integração PloSys-Granlog: aplicação da análise de granulosidade na exploração do paralelismo OU, *XXV Conferência Latino Americana de Informática (CLEI)* (Asuncion, Paraguay, August-September 1999) 911-922.

[14] K. A. M. Ali and R. Karlsson, The Muse Or-parallel Prolog Model and its Performance, *Proceedings of the North American Conference on Logic Programming* (1990) 757-776.

[15] E. Lusk et al, The Aurora Or-parallel Prolog System, *New Generation Computing* **7** (1990) 243-271.

[16] R. Rocha and F. Silva and V. Santos Costa, *YapOr: an Or-Parallel Prolog System based on Environment Copying* (LIACC, Universidade do Porto, Technical Report DCC-97-14, December 1997).

[17] M. V. Hermenegildo and K. Greene, &-Prolog and its Performance: Exploiting Independent And-Parallelism, *Proceedings of the Seventh International Conference on Logic Programming* (June 1990) 253-268.

[18] K. Shen, Overview of DASWAM: Exploitation of Dependent And-parallelism, *The Journal of Logic Programming* **29** 1996.

[19] J. Montelius, *Penny - A Parallel Implementation of AKL* (Swedish Institute for Computer Science, May 1997).

[20] R. Moolenaar and B. Demoen, A Parallel Implementation for AKL, *Proceedings of the Programming Language Implementation and Logic Programming* (1993) 246-261.

[21] K. Ueda and M. Morita, A New Implementation Technique for Flat GHC, *Proceedings of the Seventh International Conference on Logic Programming* (June 1990) 3-17.

[22] P. L. Garcia and M. V. Hermenegildo and S. K. Debray, Towards Granularity Based Control of Parallelism in Logic Programming, *International Symposium on Parallel Computation* (September 1994).

[23] P. Cousot and R. Cousot, Abstract Interpretation and Application to Logic Programs *The Journal of Logic Programming* **13** (1992) 103-179.

[24] J. Bevemyr and T. Lindgren and H. Millroth, Reform Prolog: The Language and its implementation, *Proceedings of the Tenth International Conference on Logic Programming* (June 1993) 283-298.

[25] I. C. Dutra, *Distributing And- and Or-Work in the Andorra-I Parallel Logic Programming System* (University of Bristol, PhD Thesis, February 1995).

[26] J. S. Conery, *The AND/OR Process Model for Parallel Execution of Logic Programs* (University of California, Irvine, PhD Thesis, 1983).

[27] G. Gupta and E. Pontelli and M. V. Hermenegildo, &-ACE: A High Performance Parallel Prolog System, *Proceedings of the First International Symposium on Parallel Symbolic Computation (PASCO'94)* (1994).

[28] E. Pontelli and G. Gupta and F. Pulvirenti and A. Ferro, Automatic Compile-time Parallelization of Prolog Programs for Dependent And-Parallelism, *Proceedings of the Fourteenth International Conference in Logic Programming (ICLP)* (1997).