TELIOS: A Tool for the Automatic Generation of Logic Programming Machines

Alexandros C. Dimopoulos and Christos Pavlatos and George Papakonstantinou

Abstract In this paper the tool TELIOS is presented, for the automatic generation of a hardware machine, corresponding to a given logic program. The machine is implemented using an FPGA, where a corresponding inference machine, in application specific hardware, is created on the FPGA, based on a BNF parser, to carry out the inference mechanism. The unification mechanism is based on actions embedded between the non-terminal symbols and implemented using special modules on the FPGA.

1 Introduction

Knowledge engineering approaches have extensively been used in many application domains such as medicine, scheduling and planning, control, artificial intelligence [12] etc. The low power requirements, small dimensions, and real-time limitations, which are usually specified in such applications, impose the need of designing specialized embedded systems for their implementation [13]. Therefore, the possibility of exploiting knowledge engineering approaches in embedded systems, is of crucial importance.

The first machine introduced for the implementation of logic programs (PROLOG) was the Warren Abstract Machine (WAM) [2]. The 5^{th} generation computing era was targeted towards this direction [1]. The cost for the implementation of such systems, along with their size, prevented their use in small scale applications in embedded system environments [13].

The effort of designing hardware capable of supporting the declarative programming model, for logic derivations, can now lead to intelligent embedded designs

Alexandros C. Dimopoulos · Christos Pavlatos · George Papakonstantinou
National Technical University of Athens, School of Electrical and Computer Engineering,
Iroon Polytechneiou, Zografou 15773, Athens, Greece,
e-mail: {alexdem,pavlatos,papakon}@cslab.ece.ntua.gr

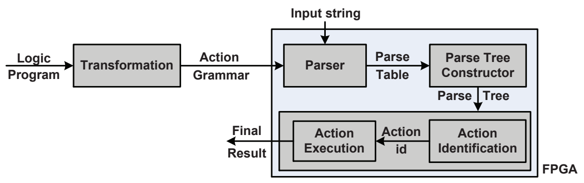

Fig. 1 Overview of our approach

which are considerably more efficient compared to the traditional ones. Some efforts have been done in the past, towards this direction [4], [11], [6] . In [4] a hardware parser was presented based on the CYK parsing algorithm. In [11] another hardware parser was presented based on the Earley's parallel algorithm [7]. Both parsers have been implemented using FPGAs. In [6] a similar approach to the one proposed here was presented. Nevertheless, the unification mechanism was implemented using softcore general purpose on chip processors, hence reducing drastically the speed up obtained by using the hardware parser.

In this paper the tool TELIOS (Tool for the automatic gEneration of LogIc prOgramming machineS) is presented. The user describes his logic program in a subset of PROLOG and the systems generates the necessary code to be downloaded to an FPGA (Field Programmable Gate Array). This FPGA is the machine for this specific logic program. The proposed implementation follows the architecture shown in Fig. 1. The given logic program can be transformed to an equivalent grammar, which feeds the proposed architecture, in order the different components to be constructed. The contribution of this paper is:

1. The modification of the hardware parser of [11], in order to be used for logic programming applications. It is noted that the parser of [11] is two orders of magnitude faster than the one used in [6].
2. The (automatic) mapping of the unification mechanism, to actions, easily implementable in FPGAs. To the best of the authors knowledge, this is the first effort to implement logic programs on FPGAs, without the use of an external real processor or a softcore one on the same chip.

2 Theoretical Background

Attribute Grammars (AG) [8] have been extensively used for logic programming applications [10], [5], [9]. The basic concepts for transforming a logic program to an equivalent AG are the following: Every inference rule in the initial logic program can be transformed to an equivalent syntax rule consisting solely of non-terminal symbols. Obviously, parsing is degenerate since there are no terminal symbols. For every variable existing in the initial predicates, two attributes are attached to the corresponding node of the syntax tree, one synthesized and one inherited. Those attributes assist in the unification process of the inference engine. For more details the user is referred to [10], [9]. The computing power required for the transformation of logic programs to AGs is the one of L-attributed AGs [8]. In these grammars the attributes can be evaluated traversing the parse tree from left to right in one pass.

In this paper it is shown that L-attributed AGs are equivalent to "action" grammars, which are introduced in this paper, due to their easy implementation in hardware. Hence, we can transform a logic program to an equivalent action grammar.

The Action Grammars, are defined in this paper as BNF grammars, augmented with "actions". Actions are routines which are executed before and after the recognition of an input substring corresponding to a non-terminal. In the rule: $< NT >::= \ldots [A_i] < NT_i > \ldots \ \ < NT_j > [A_j]$, the actions to be taken are the execution of the routine A_i before recognizing the non-terminal NT_i and the execution of the routine A_j after the recognition of the non-terminal NT_j. The execution of A_i and A_j takes place after the generation of all possible parse trees. In the case of Earley's algorithm this is done in parallel, so that at the end of the parsing process all possible parse trees are available.

As it was stated before, it will be shown here that action grammars are equivalent to L-attributed grammars. For this purpose, some rules must be applied: 1) For each attribute (synthesized or inherited) a stack is defined, having the same name as the attribute. 2) At the end of each rule, unstacking of the synthesized attributes, of the descendant (children nodes) of the non-terminal at the left hand side of the rule (parent node), is done. These synthesized attributes are at the top of the stack. The synthesized attribute of the parent node is calculated according to the corresponding semantic rule and is pushed to the appropriate stack as shown in Fig. 2a. In this way, it is sure that at the top of the stack, the synthesized attributes of the children nodes of the parent (up to the corresponding child) are placed in sequence. 3) Regarding the inherited attributes: a) A push is done at the corresponding stack of the inherited attribute, the first time it is evaluated (produced). A pop is done at the time the inherited attribute is needed (consumed) as in Fig. 2b. b) If in a rule an inherited attribute is used in more than one children non-terminals (as in Fig. 2b), then the same number of pushes of that attribute should be done. c) If a value transfer semantic rule (for the same attribute) is needed in the AG, then no action is required for inherited and synthesized attributes (as in Fig. 2b). In Fig. 2, i is an inherited attribute, s a synthesized, x_i auxiliary (temporary) variables and the arrows indicate attribute dependencies.

The rules described above, will be further clarified with an example which follows.

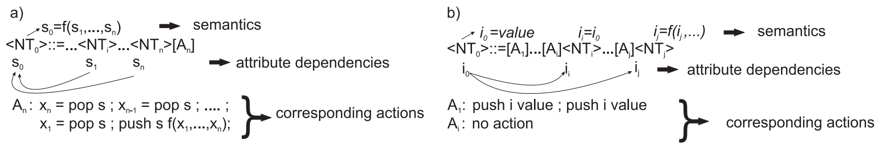
a)
s0=f(s1,...,sn)
semantics
<NT0>::=...<NTi>...<NTn>[An]
s0 s1 sn
attribute dependencies
An: xn = pop s ; xn-1 = pop s ; .... ;
x1 = pop s ; push s f(x1,...,xn);
corresponding actions
b)
i0=value ii=i0 ij=f(ij,...)
semantics
<NT0>::=[A1]...[Ai]<NTi>...[Aj]<NTj>
i0 ii ij
attribute dependencies
A1: push i value ; push i value
Ai: no action
corresponding actions

Fig. 2 a) Synthesized attribute example b) Inherited attribute example

3 An Illustrative Example

In order to clarify the aforementioned transformation, we demonstrate a toy-scale example of a logic program which is transformed to its action grammar equivalent one. Consider that we have the knowledge base (logic program) illustrated in Table 1 (First Column) and we want to ask the question “p is successor of whom?” i.e. $\mathit{Successor}(p,?)$. The syntax rules, which form the equivalent action grammar evaluator, are illustrated in Table 1 (Second Column) along with the definition of the actions. The equivalent action grammar does not contain any terminal symbols, therefore every fact P(x,y) is transformed to a syntactic rule of the form P $\rightarrow$ d, where d is a dummy symbol that is also used for the representation of the empty input string. The meta-variable *flag* arises from the transformation of the logic program to the equivalent AG. Its value is used by the attribute evaluator to discard useless subtrees, when it is equal to zero.

It is noted that we have four attributes, two for the two parameters of each predicate, and two (one inherited and one synthesized) for each parameter, denoted by P_{qr}. P_{qr} stands for parameter q of the predicates ($q \in \{1,2\}$ in our example) and r $\in \{$i,s$\}$ where i means inherited and s synthesized attribute. Hence, in our example we have the attributes P_{1i}, P_{2i}, P_{1s} and P_{2s}. For each attribute a stack is kept i.e. $stack_{1i}$, $stack_{2i}$, $stack_{1s}$ and $stack_{2s}$, respectively.

The question asked has two solutions, which are “j” and “b”. The corresponding parse tree, decorated with the actions are illustrated in Fig. 3. A tracing of the execution of the actions (A_0, A_1, A_3, A_{11}, A_4, A_6, A_8, A_7, A_5, A_2 and A_0, A_1, A_6, A_{11},

Table 1 An AG equivalent representation of the knowledge based of the “successor problem”

Informal Definition of the Knowledge Base	**Equivalent Action Grammar**
1. Goal(X,Y) $\leftarrow$ Successor(X,Y)	$< \mathit{Goal} > ::= [A_1] < \mathit{successor} > [A_2]\$$
2. Successor(X,Y) $\leftarrow$ Parent(Z,X) and Successor(Z,Y)	$< \mathit{successor} > ::= [A_3] < \mathit{parent} > [A_4] < \mathit{successor} >[A_5]\$$
3. Successor(X,Y) $\leftarrow$ Parent(Y,X)	$< \mathit{successor} > ::= [A_6] < \mathit{parent} > [A_7]\ \$$
4. Parent(j,b)	$< \mathit{parent} >::= d\ [A_8]\ \$$
5. Parent(j,l)	$< \mathit{parent} >::= d\ [A_9]\ \$$
6. Parent (b,a)	$< \mathit{parent} >::= d\ [A_{10}]\ \$$
7. Parent (b,p)	$< \mathit{parent} >::= d\ [A_{11}]\ \$$
[A_1]$\rightarrow$ no action	
[A_2]$\rightarrow$ no action	
[A_3] $\rightarrow$ tmp_1 = pop ($stack_{1i}$);push($stack_{2i}$,tmp_1);push ($stack_{1i}$,nil)	
[A_4] $\rightarrow$ tmp_1 = pop ($stack_{1s}$);push($stack_{1i}$,tmp_1);	
[A_5]$\rightarrow$ tmp_1=pop($stack_{2s}$); tmp_2=pop($stack_{2s}$);tmp_3=pop($stack_{1s}$);push($stack_{2s}$,tmp_1); push($stack_{1s}$,tmp_2);	
[A_6] $\rightarrow$ tmp_1 = pop ($stack_{2i}$); tmp_2 = pop ($stack_{1i}$); push ($stack_{1i}$, tmp_1); push ($stack_{2i}$, tmp_2);	
[A_7] $\rightarrow$ tmp_1 = pop ($stack_{2s}$); tmp_2 = pop ($stack_{1s}$); push ($stack_{1s}$, tmp_1); push ($stack_{2s}$, tmp_2);	
[A_8] $\rightarrow$ tmp = pop ($stack_{1i}$); if ((tmp != nil) and (tmp != “j”)) then flag =0 else push ($stack_{1s}$, “j”); tmp = pop ($stack_{2i}$); if ((tmp != nil) and (tmp != “b”)) then flag =0 else push ($stack_{2s}$, “b”) ;	
[A_9] $\rightarrow$ tmp = pop ($stack_{1i}$); if ((tmp != nil) and (tmp != “j”)) then flag =0 else push ($stack_{1s}$, “j”) ; tmp = pop ($stack_{2i}$); if ((tmp != nil) and (tmp != “l”)) then flag =0 else push ($stack_{2s}$, “l”) ;	
[A_{10}] $\rightarrow$ tmp = pop ($stack_{1i}$); if ((tmp != nil) and (tmp != “b”)) then flag =0 else push ($stack_{1s}$, “b”) ; tmp = pop ($stack_{2i}$); if ((tmp != nil) and (tmp != “a”)) then flag =0 else push ($stack_{2s}$, “a”) ;	
[A_{11}] $\rightarrow$ tmp = pop ($stack_{1i}$); if ((tmp != nil) and (tmp != “b”)) then flag =0 else push ($stack_{1s}$, “b”) ; tmp = pop ($stack_{2i}$); if ((tmp != nil) and (tmp != “p”)) then flag =0 else push ($stack_{2s}$, “p”) ;	
Goal (p,x)	$< \mathit{Goal} > ::= [A_0]\ \$$
[A_0] $\rightarrow$ push ($stack_{1i}$, p);push ($stack_{2i}$, nil);	

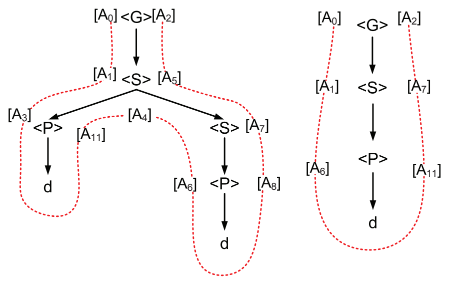

Fig. 3 Parse trees for the "successor" example leading to solutions (Note that tree traversal is top-bottom, left to right)

A_7, A_2 for the two parse trees) will leave at the top of the stack stack_{2s} the values "j", "b" respectively. The predicate names have been abbreviated.

4 Implementation

Chiang & Fu [3] parallelized Early's parsing algorithm [7], introducing a new operator $\otimes$ and proposed a new architecture which requires $\frac{n*(n+1)}{2}$ processing elements (PEs) for computing the parse table. A new combination circuit was proposed in [11] for the implementation of the $\otimes$ operator. In this paper a modification of the parsing algorithm of [11] has been done in order to compute the elements of the parse table PT by the use of only n processing elements that each one handled the cells belonging to the same column of the PT.

It is obvious that since parsing is top-down, when recursion occurs and no input string is used (the empty string is the input string), we may have infinite creation of dotted recursion rule in the boxes. Hence, we have to predefine the maximum recursion depth as well as the maximum number of the input characters (d characters) in the input string, as installation parameters. The unification mechanism has been implemented through actions. The parse trees are constructed from the information provided by the parser. Actions are identified in the Action Identification module and executed in the Action Execution module (Fig. 1).

The system TELIOS has been implemented in synthesizable Verilog in the XILINX ISE 8.2[1] environment while the generated source has been simulated for validation, synthesized and tested on a Xilinx SPARTAN 3E FPGA. Furthermore, it has been tested with hardware examples we could find in the bibliography and in all cases our system runs faster. In the case of the well-documented "Wumpus World Game" and of finding paths in a directed acyclic graph [6], our system was two orders of magnitude faster than the one of [6] required.

[1] Xilinx Official WebSite, http://www.xilinx.com

5 Conclusion and Future Work

The system is very useful in cases where rapid development of small scale intelligent embedded hardware has to be used in special purpose applications, locally in dangerous areas, in robotics, in intelligent networks of sensors e.t.c.. The system in its present form accepts a subset of PROLOG e.g. only variables and constants as parameters of the predicates. Nevertheless, since we have shown the equivalence of L-attributed grammars with action grammars and L-attributed grammars can cover many other characteristics of PROLOG [9] (e.g. functors), it is straightforward to extend our system. Future work aims at: 1) Combining the modules of parsing and action execution in one module so that parsing will be completely semantically driven. This will solve the recursion problem in a more efficient way. 2) Extending the power of the grammar from L-attributed to many passes ones. 3) Applying the tool in medical applications. 4) Extending the PROLOG subset used in this paper.

Acknowledgements This work has been funded by the project PENED 2003. This project is part of the OPERATIONAL PROGRAMME ”COMPETITIVENESS” and is co-funded by the European Social Fund (80%) and National Resources (20%).

References

1. *Communications of the ACM*, 26(9), 1983.
2. Hassan Ait-Kaci. *Warren's abstract machine : a tutorial reconstruction*. MIT Press, 1991.
3. Y T Chiang and King-Sun Fu. Parallel parsing algorithms and VLSI implementations for syntactic pattern recognition. *IEEE Trans. on PAMI*, 6:302–314, 1984.
4. C. Ciressan, E. Sanchez, M. Rajman, and J.C. Chappelier. An fpga-based coprocessor for the parsing of context-free grammars. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 236, Washington, DC, USA, 2000.
5. Pierre Deransart, Bernard Lorho, and Jan Maluszynski, editors. *Proceedings of the 1st International Workshop on Programming Language Implementation and Logic Programming, PLILP'88, Orléans, France, May 16-18, 1988*. Springer, 1989.
6. A. Dimopoulos, C. Pavlatos, I. Panagopoulos, and G. Papakonstantinou. An efficient hardware implementation for AI applications. *Lecture Notes in Computer Science*, 3955:35–45, April 2006.
7. Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970.
8. Jukka Paakki. Attribute grammar paradigms a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, 1995.
9. T. Panayiotopoulos, G. Papakonstantinou, and G. Stamatopoulos. Ai-debot paper. *Angewandte Informatik*, 88(5), 1988.
10. G Papakonstantinou, C Moraitis, and T Panayiotopoulos. An attribute grammar interpreter as a knowledge engineering tool. *Angew. Inf.*, 28(9):382–388, 1986.
11. C. Pavlatos, A. C. Dimopoulos, A. Koulouris, T. Andronikos, I. Panagopoulos, and G. Papakonstantinou. Efficient reconfigurable embedded parsers. *Computer Languages, Systems & Structures*, 35(2):196 – 215, 2009.
12. Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.
13. Frank Vahid and Tony Givargis. *Embedded System Design: A Unified Hardware/Software Introduction*. WILEY, 2002.