An Efficient Hardware Implementation for AI applications

Alexandros Dimopoulos, Christos Pavlatos, Ioannis Panagopoulos, and George Papakonstantinou

National Technical University of Athens, Dept. of Electrical and Computer Engineering
Zographou Campus, 157 73 Athens, Greece
{alexdem, pavlatos, ioannis, papakon}@cslab.ece.ntua.gr

Abstract. A hardware architecture is presented, which accelerates the performance of intelligent applications that are based on logic programming. The logic programs are mapped on hardware and more precisely on FPGAs (Field Programmable Gate Array). Since logic programs may easily be transformed into an equivalent Attribute Grammar (AG), the underlying model of implementing an embedded system for the aforementioned applications can be that of an AG evaluator. Previous attempts to the same problem were based on the use of two separate components. An FPGA was used for mapping the inference engine and a conventional RISC microprocessor for mapping the unification mechanism and user defined additional semantics. In this paper a new architecture is presented, in order to drastically reduce the number of the required processing elements by a factor of *n* (length of input string). This fact and the fact of using, for the inference engine, an extension of the most efficient parsing algorithm, allowed us to use only one component i.e. a single FPGA board, eliminating the need for an additional external RISC microprocessor, since we have embedded two "PicoBlaze" Soft Processors into the FPGA. The proposed architecture is suitable for embedded system applications where low cost, portability and low power consumption is of crucial importance. Our approach was tested with numerous examples in order to establish the performance improvement over previous attempts.

1 Introduction

Although Artificial Intelligence (AI) has already been a challenging research area for more than 50 years, it still remains one of the most modern and interesting fields. Knowledge engineering and logic programming approaches have extensively been used in a considerable number of application domains, which range from medicine to game theory [1]. It's common for various research areas to resort in AI techniques, seeking for intelligent tools to enhance their performance. On the other hand, techniques from other research fields can be embedded into AI applications. Such an approach is reported in the present paper, in which we show how hardware/software co design techniques can be exploited, so as to map AI application on a single FPGA (Field Programmable Gate Array) board. Since most AI applications need to conform to very strict real-time margins, one of the key requirements for the efficiency of such

systems is that of performance. As a result, designing fast algorithms for logic derivations is a key requirement for the efficiency of the implementation of an intelligent embedded system.

It is well known that knowledge representation and processing can be accomplished by two approaches, the declarative and the procedural one. Since Attribute Grammars (AGs) [2] can easily integrate the two approaches in a single tool, this approach appears to be ideal [3], [4], [5], to model AI applications and specifically PROLOG logic programs [6]. Moreover, the field of AGs' processing is fairly mature and many efficient implementations of compilers and interpreters for such evaluation processes can be utilized.

AGs were introduced in 1968 by Knuth [2]. The addition of attributes and semantic rules to Context Free Grammars (CFGs) augmented their expressional capabilities, making them in this way a really useful tool for a considerable number of applications. AGs have extensively been utilized in AI applications [3], [4], [5], [7], [8] structural pattern recognition [9], [10], compiler construction [11], and even text editing [12]. However, the additional complexity imposed by the added characteristics, along with the need for fast CF parsing by special applications, dictates the parallization of the whole procedure (parsing and attribute evaluation) as an attractive alternative to classical solutions.

In this paper we present a new hardware implementation for AI applications, based on AGs. We have improved previous approaches by reducing the number of required processing elements by a factor of n (length of input string). This fact allowed us to use only one component i.e. a single FPGA board, eliminating the need for an external microprocessor, as presented in previous works [7], [8], [13], [14], [15]. Additionally the attribute evaluation algorithm – that implements the unification mechanism and user defined additional semantics – has been improved as well and has been divided into two parts that are executed simultaneously into two processors. Both processors are mapped on the same Xilinx Spartan-II FPGA board, together with the inference engine. Consequently the unification process and the inference mechanism are executed on the same component, an FPGA board. Therefore the proposed architecture is suitable for embedded system applications where low cost, portability and low power consumption is of crucial importance. The downloaded processors, responsible for the attribute evaluation process, are two "PicoBlaze Soft Processor" [16] provided by Xilinx. The PicoBlaze Soft Processor is a very simple 8-bit micro controller designed to be 100% embedded into devices such as the Spartan-II we used. The processors interface with the parser using hardware/software co design methods (see Fig.1), while all data are stored in a shared by all components RAM.

Our approach has been simulated for validation, synthesized and tested on a Xilinx Spartan-II FPGA board, with numerous examples in order to establish the performance improvement over previous attempts. The performance speed up is application depended, i.e. on the length of the produced AG. Our contribution in this work is summarized as follows:

- We improved the parallel parsing architecture by eliminating the required processing elements by a factor of *n* (input string length) for the subset of AGs produced by PROLOG logic programs.

- We divided the attribute evaluation process into two pieces so as to be executed in parallel on two separate processors, concurrently with the parsing task.
- We mapped the whole implementation (two processors, parser, RAM) into a single component (FPGA).

The rest of the paper is organized as follows. In Section 2, the necessary theoretical background is presented. In Section 3, the implementation details are analyzed, while in Section 4, an illustrative example is demonstrated and performance evaluation is discussed. Finally, Section 5 concludes and presents our future work.

2 Theoretical Background

In this section we give the necessary fundamental definitions and a brief description of how PROLOG logic programs can be transformed into AGs. We will not explain in details theoretical issues, trying to focus on architectural aspects.

An AG is based upon a CFG. A CFG is a quadruple G = (N, T, R, S), where N is the set of non-terminal symbols, T is the set of terminal symbols, R is the set of syntactic rules, written in the form $A \longrightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$) and S is the start symbol. We use capital letters A, B, C… to denote non terminal symbols, lowercases a, b, c… to denote terminal symbols and Greek lowercases and α, β, γ... for $(N \cup T)^*$ strings. An AG is a quadruple AG = {G, A, SR, d} where G is a CFG, $A = \cup A(X)$ where A(X) is a finite set of attributes associated with each symbol $X \in V$. Each attribute represents a specific context-sensitive property of the corresponding symbol. The notation X.a is used to indicate that attribute a is an element of A(X). A(X) is partitioned into two disjoint sets; the set of synthesized attributes $A_s(X)$ and the set of inherited attributes $A_i(X)$. Synthesized attributes X.s are those whose values are defined in terms of attributes at descendant nodes of node X of the corresponding semantic tree. Inherited attributes X.i are those whose values are defined in terms of attributes at the parent and (possibly) the sibling nodes of node X of the corresponding semantic tree. Each of the productions $p \in R$ ($p: X_0 \longrightarrow X_1 \ldots X_k$) of the CFG is augmented by a set of semantic rules SR(p) that defines attributes in terms of other attributes of terminals and on terminals appearing in the same production. The way attributes will be evaluated depends both on their dependencies to other attributes in the tree and also on the way the tree is traversed. Finally d is a function that gives for each attribute a its domain d(a).

In [4], [5] an effective method based on Floyd's parser [17] was presented that transforms any initial logic programming problem to its attribute grammar equivalent representation. The basic concepts underlying this approach are the following: every logic rule in the initial logic program can be transformed to an equivalent syntax rule consisting solely of non-terminal symbols. The general idea of using an AG for knowledge representation is to use only one terminal symbol, the NULL symbol. Thus, the grammar recognizes only empty strings of characters. During the recognition of an empty string the semantics can be such that at the time they are evaluated they accomplish the inference required. For example: $R_0(\ldots) \leftarrow R_1(\ldots) \wedge \ldots \wedge R_m(\ldots)$ is transformed to the syntax rule: $R_0(\ldots) \longrightarrow R_1 \ldots R_m|.$ ("|." represents the end of the rule and "|" represents logic OR). Finally facts of the

inference rules are transformed to terminal leaf nodes of the syntax tree referring to the empty string. For example the facts: R_g(a,b), R_g(c,d), R_g(e,f) are transformed to: $R_g \longrightarrow$ ||||. 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. The attribute evaluation rules are constructed based on the initial logic program. A detailed methodology for specifying those rules can be found in [5]. Attributes at the leaf nodes of the tree are assigned values from the constants in the facts of the logic program. The inference process is carried out during tree derivations and a function is evaluated at the insertion/visit of each node that computes the attribute rules performing the unification procedure. The way knowledge representation can be accomplished using AGs is illustrated in the example of Sec. 4.

3 The proposed Implementation

3.1 Overview of our approach

In this paper the underlying model of implementing an embedded system for AI applications is that of an AG evaluator. The AG evaluation process is usually divided into two discrete tasks, that of syntactic parsing and that of semantic evaluation. The first corresponds to the inference engine, while the second to the unification mechanism. In the proposed embedded system, the inference engine is implemented using the hardware parsing architecture presented in [13], applying the necessary modifications analyzed in 3.2. The unification mechanism is carried out by the use of two processors embedded in the same FPGA with the parser. The whole process is controlled by the Control Unit, while all data are stored and retrieved by all components in a shared RAM. Our architecture is illustrated in Fig. 1 and analytically presented in the next sections.

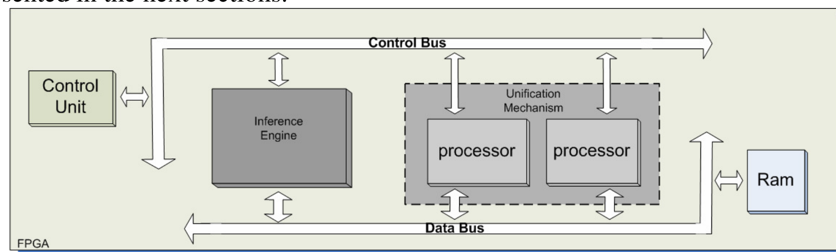


Fig. 1. The proposed architecture

3.2 The Inference Engine (Hardware Parser)

As referred in Sec. 2, every logic rule or fact corresponds to a syntactic rule. The set of these rules produces a CFG, which should be syntactically recognized. Hence, the inference task is carried out by a parser. The underlying algorithm of the parser is

based on the most efficient parsing algorithm [18] in a parallel version presented by Chiang & Fu [14].

The basic innovation of the top-down parser that Earley [18], was the introduction of a symbol called dot "•" that does not belong to the grammar. The utility of the dot in a rule (now called dotted rule) is to separate the right part of the rule into two subparts. For the subpart at the left of the dot, it has been verified that it can generate the input string examined so far. However, for the subpart at the right of the dot, it still remains to check whether or not it can generate the rest of the input string. The algorithm scans the input string $a_1a_2a_3…a_n$ from left to right (where n is the input string length). As each symbol a_i is scanned, a set S_i of states is constructed which represents the condition of the recognition process at the point in the scan. A state is a 3-tuple $\{r, l, f\}$ where r is the number of the rule, l is the position of the dot and f is the set that the state was first created.

In 1980 Graham et al [19] proposed the use of an array PT (Parse Table) instead of Earley's set structure. The element of the array pt(i,j) contains all the dotted rules that belong to set S_j and were firstly created in set S_i. Particularly the jth column of the array PT corresponds to set S_j. Only the elements on or above the diagonal are used.

Chiang & Fu proved that the construction of the parsing table can be parallelized with respect to n by computing, in parallel, at every step k the cells pt(i,j) for which $j\text{-}i=k\geq1$. The architecture they proposed needs $n^2/2$ processing elements that each one computes the states of a cell of array PT. In every execution step (te_1, te_2, … te_n) each processor computes one cell and then transmits this cell to others processors as shown in Fig. 2(a). Chiang & Fu also introduced a new operator ⊗. Every cell pt(i,j) is a set of dotted rules (states) that can be calculated by the use of this operation ⊗, the cells of the same column and the cells of the same row as shown in equation 1.

An enhanced version of Chiang & Fu architecture was presented in [13] that computed the elements of the PT by the use of only n processing elements that each one handled the cells belonging to the same column of the PT, as shown in Fig. 2(b).

The general idea of using an AG for knowledge representation is to use only one terminal symbol, the NULL symbol. Thus, the grammar recognizes only empty strings of symbols. During the recognition of an empty string (actually the empty string) the semantics can be such that at the time they are evaluated they accomplish the inference required. In order to make the grammar compatible with the chosen parser, we introduce the use of a dummy terminal symbol "d". Consequently, the parser recognizes inputs strings of the form "dd…d|.". The length of the input is problem length depended. Since a_i=d for $1\leq i\leq n$, the cells that are executed during execution step te_1, as shown in equation 1 are equal to pt(i,j) = pt(i,j -1) ⊗ d. However, the cells that belong to the main diagonal are the same syntax-wise. Therefore, all the cells that are executed during execution step te_1 i.e. pt(i,j), $1\leq i<n$, $j=i+1$, are the same. Inductively, based on that critical comment and due to the form of equation 1, it can easily be proven that all the cells pt(i,j) that belong to the same diagonal contain the same states.

It must be clarified that although the cells may have the same states, the values of the attributes are clearly different, since the attributes are strictly connected to their position in the parse table and to the values of the attributes of the predecessor and successor symbols.

$$pt(i,j)=\begin{cases} scanner: & pt(i,j-1)\otimes a_j \quad \cup \\ completer: & pt(i,i+1)\otimes pt(i+1,j) \quad \cup \\ & pt(i,i+2)\otimes pt(i+2,j) \quad \cup \\ & \dots \quad \cup \\ & pt(i,j-1)\otimes pt(j-1,j) \end{cases} \tag{1}$$

Thus, the parsing task can be accomplished by the use of one processing element, instead of *n*, that computes only the cells of the first row of the PT, as shown in Fig. 2(c). Once a cell is calculated, it is replicated to the others of the same diagonal so as to fill, the necessary for the attribute evaluation, PT. For example pt(0,1) will be copied to pt(1,2) and pt(2,3). The overhead for this transition is negligible relatively to the overall procedure. The architecture of the one parsing element follows the one presented in [13] achieving a speed-up by a factor of approximately 5, compared to software approaches. Additionally, the fact that we should compute the cells that belong only to the first row, augments drastically the speed-up. As the input string length and therefore the PT size increases, the speed-up increases as well. Experimental results are given in the next section.

The reduction of the required parsing processing elements simplifies the design allowing us to incorporate the processors responsible for the Attribute Evaluation into the same FPGA board, eliminating the need for an external microprocessor.

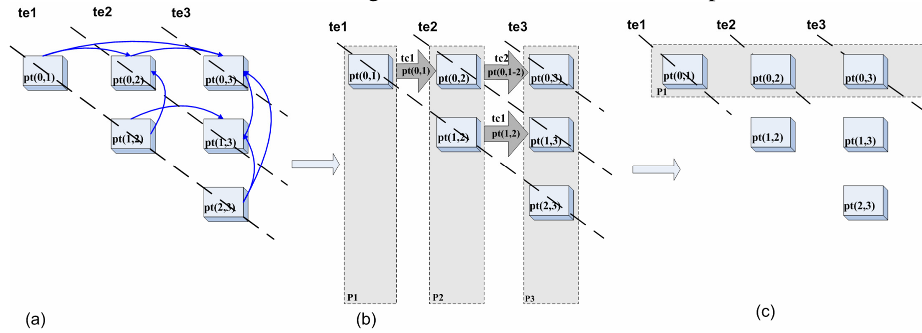


Fig. 2. (a) Chiang & Fu's parallel architecture (n=4) (b) Parsing Architecture for Grammar with Terminal Symbols (c) Parsing Architecture for Grammar without Terminal Symbols

3.3 The Unification Mechanism (Attribute Evaluator)

Once the parser has completed the computation of a PT cell pt(0,j), the attribute evaluation process may begin –evaluating the jth column– concurrently with the parser that computes the next cell pt(0,j+1).

In order to compute the inherited attributes of a state ($state_{current}$) in some cases, data from two other states ($state_1$ and $state_2$) are needed; one from the same row and one from the same column. The state from the same column may be placed either in the same cell or in one bellow.

To face both abovementioned cases, the way the column should be traversed is from bottom to top in relation to the cells and top to bottom in relation to the states inside each cell. Due to the nature of Earley's parsing algorithm (top-down, left to right) synthesized attributes may be evaluated correctly with solely the data that have

already been transferred there. This action takes place when the dot symbol "•" reaches the end of the rule.

The attribute evaluation takes place in the PicoBlaze Soft Processors. The PicoBlaze Soft Processor is a very simple 8-bit micro controller designed to be 100% embedded into Spartan-II device. The PicoBlaze Soft Processor features 16 general purpose registers. A simple ALU supporting ADD/SUB, logical, shifts and rotates, conditional jumps and nested subroutine calls.

In the proposed implementation we divided the attribute evaluation process into two parts, so as to be evaluated to two separate processors in an attempt to increase the performance. Since the attribute evaluation of a column in processor1 completes to the point that the evaluation of the next column may start, processor1 sends an interrupt to processor2 to notify it that it may start. Then processor2 handles the evaluation of the next column and so on, as shown in Fig.3. In Fig. 3, it is clearly shown how our approach outperforms the conventional one, mainly due to the three following factors:

- The parsing is carried out in hardware and consequently is completed in shorter time.
- The attribute evaluation is taking place concurrently with the parsing task and not sequentially after the computation of the whole PT.
- The burden of the attribute evaluation is handled by two processors, reducing the time required, due to the pipeline parallelization.

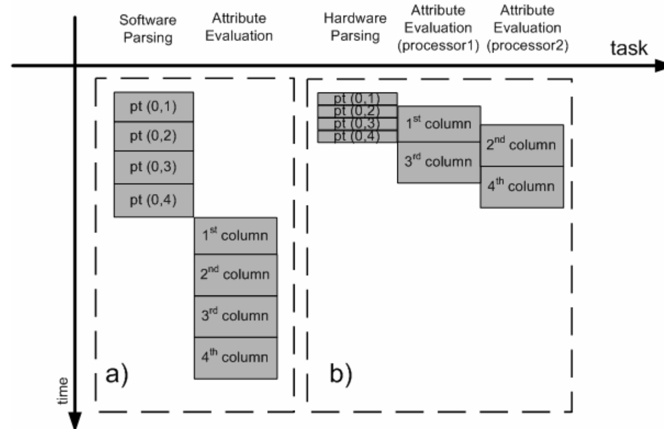


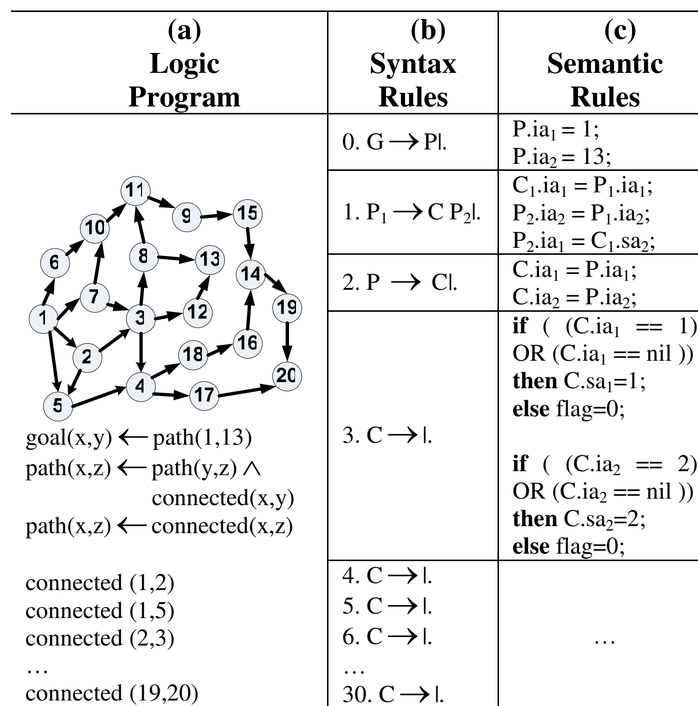
Fig. 3. Comparison of our approach (b) against the software approach (a)

4 An Illustrative Example

The way knowledge representation can be accomplished using AGs is illustrated in the following example. Consider the case where an application needs to find whether a path exists in a directed acyclic graph (Table 1) between two nodes of the graph and if so how many such paths exist. For a graph of k nodes with each node represented by a number i, where $0<i<k$ we define the predicate connected(i,j) which is true whenever there is a directed edge leading from i to j. A simple logic program, for

finding paths from an arbitrary node x to another node z in the directed acyclic graph, is provided in Table 1(a). The equivalent attribute grammar syntax rules handling this inference procedure are provided in Table 1 (b) and the attribute evaluation rules for the unification process are shown in Table 1 (c). In the syntax rules the goal is represented by “G”, path by “P” and connected by “C”. Let’s assume that the goal connection is from 1 to13.

Table 1.(a) Directed acyclic graph and Logic Program for finding a path in a directed acyclic graph (b) Equivalent syntax rules for the attribute grammar to be used as inference engine (c) Semantic Rules



(a) Logic Program	**(b) Syntax Rules**	**(c) Semantic Rules**
	0. $G \rightarrow P\vert.$	$P.ia_1 = 1$; $P.ia_2 = 13$;
	1. $P_1 \rightarrow C\ P_2\vert.$	$C_1.ia_1 = P_1.ia_1$; $P_2.ia_2 = P_1.ia_2$; $P_2.ia_1 = C_1.sa_2$;
	2. $P \rightarrow C\vert.$	$C.ia_1 = P.ia_1$; $C.ia_2 = P.ia_2$;
goal(x,y) ← path(1,13) path(x,z) ← path(y,z) ∧ connected(x,y) path(x,z) ← connected(x,z)	3. $C \rightarrow \vert.$	**if** (($C.ia_1$ == 1) OR ($C.ia_1$ == nil)) **then** $C.sa_1$=1; **else** flag=0; **if** (($C.ia_2$ == 2) OR ($C.ia_2$ == nil)) **then** $C.sa_2$=2; **else** flag=0;
connected (1,2) connected (1,5) connected (2,3) … connected (19,20)	4. $C \rightarrow \vert.$ 5. $C \rightarrow \vert.$ 6. $C \rightarrow \vert.$ … 30. $C \rightarrow \vert.$	…

Provided that the technology used for the hardware implementation is the same for both the FPGA and the microprocessor (if we run the application using a prolog program on a conventional microprocessor) we can safely use the number of the required clock cycles as measure of the efficiency of the two approaches (hardware versus software). Additionally, the computational power of processors used in embedded system, is comparable to that of an FPGA. Hence the performance in all implementations is measured using the number of the required clock cycles, so as to purely compare the architecture, regardless of the technology used. The clock cycles in the software implementations refer to those needed by the processor to execute the algorithm.

In Table 2 measurements are presented for both the software and the hardware approach. Specifically, we have taken individual measurements for i) The software Parser, ii) The Hardware Parser (computation of the first row), iii) The Hardware

Parser including the transmission process (filling all the PT), iv) The Attribute Evaluation using only one processor (Pentium II 350 MHz) and v) The Attribute Evaluation using our approach with two PicoBlaze Soft Processors embedded in the Xilinx Spartan-II FPGA. Finally we present the speed-up individually for the parser, the attribute evaluation and the total speed-up (see Fig.4). Furthermore, in Fig. 5 we compare the hardware against the software approach. Unfortunately, due to the difference in magnitude, some measurements cannot appear. Mainly the hardware parser that is under the attribute evaluation (in the FPGA using the two processors).

Table 2. Measurements in clock cycles

Input String Length	4	8	12	16	20
Software Parser	13,560	49,358	115,789	223,153	381,450
Hardware Parser	4,173	9,274	12,997	17,988	25,349
Transmission	96	336	704	1,200	1,824
Hardware Parser + Transmission	4,269	9,610	13,701	19,188	27,173
Attribute Evaluation using one processor	256,342	860,578	1.565,480	2,464,523	3,629,427
Attribute Evaluation using two processors	229,687	622,222	948,842	1,286,956	1,674,223
Parsing Speed-up	3.18	5.14	8.45	11.63	14.04
Attribute Evaluation Speed-up	1.12	1.38	1.65	1.92	2.17
Software approach	269,902	909,936	1,681,269	2,687,676	4,010,877
Our approach	233,956	631,832	962,543	1,306,144	1,701,396
Final Speed-up	1.15	1.44	1.75	2.06	2.36

We can see from Table 2 and Fig. 4, 5 that although we have a very high speed-up for the hardware inference machine (hardware parser), the corresponding speed-up for the unification mechanism (attribute evaluation) is non analogous. These results were expected according to Fig.3. Hence, the overall performance is reduced due to the unification mechanism, i.e. the bottleneck is in the unification.

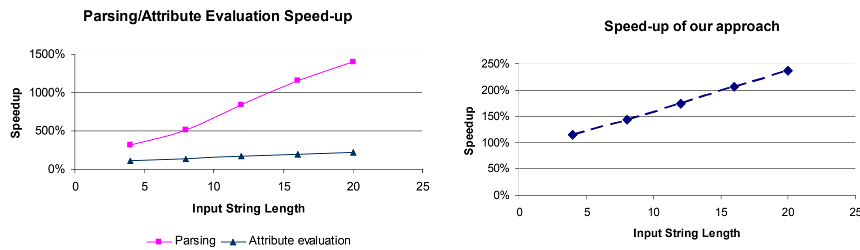


Fig. 4. (a) Parsing/Attribute Evaluation Speed-up (b) Speed-up of our approach compared against software approach

There are four solutions to the above problem. One is to use more processors embedded in the FPGA for the parallel evaluation of the semantics. The second is to

use a very fast general purpose external microprocessor for only the evaluation of the semantics. The third is to implement the semantics mapping them directly on the FPGA hardware and not through software on the microprocessor embedded on the FPGA board. The fourth solution is to choose another parallel parsing algorithm which will probably be more suitable for AGs evaluation.

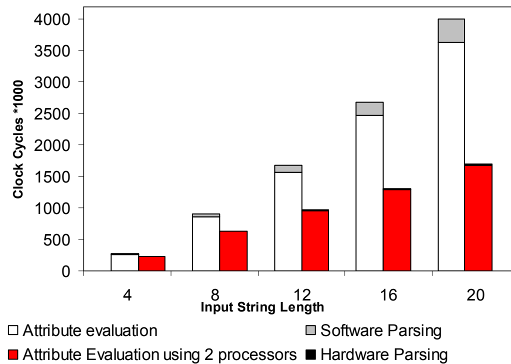


Fig. 5. Comparison of hardware against software approach

The first solution is limited due to the specific size of the FPGA, while the second one violates the requirements of small scale embedded systems which are: low cost, portability, small size, low power consumption e.t.c. The proposed architecture fulfills the above described characteristics, improving also the performance over the software solution, when we use a microprocessor of the same technology. We are currently working for implementing the third solution and we investigate the use of other parallel parsing algorithms more suitable for AGs

5 Conclusion and Future Work

In this paper we present an efficient embedded system for AI applications. The inference engine, as well as the unification mechanism is incorporated in a single FPGA. The proposed architecture is suitable for embedded system applications where low cost, portability and low power consumption is of crucial importance. Interesting enhancements have been applied to both aforementioned tasks, achieving a total speed-up that is depended on the size of the application.

This work is a part of a project[1] for developing a platform (based on AGs) in order to automatically generate special purpose embedded systems. The application area will be that Artificial Intelligence (AI) and of Syntactic Pattern Recognition for Electrocardiogram (ECG) analysis using software hardware co design techniques.

Our future research interest is to automate the whole procedure, so as to automatically map PROLOG logic programs into FPGAs. Furthermore, the speed-up

[1] This work is co - funded by the European Social Fund (75%) and National Resources (25%) - the Program PENED 2003.

would drastically increase if the attribute evaluation process was described in Hardware Description Language (HDL) and download into the FPGA.

References

1. Russel, S., Norvig P.: Artificial Intelligence, a modern approach. Prentice Hall, (1995)
2. Knuth, D.: Semantics of context free languages. Math. Syst. Theory, Vol.2, No.2, (1971) 127-145
3. Deransart, P., Maluszynski J.: A grammatical view of logic programming. MIT Press, (1993)
4. Papakonstantinou, G., Kontos J.: Knowledge Representation with Attribute Grammars. The Computer Journal, Vol. 29, No. 3, (1986)
5. Papakonstantinou, G., Moraitis, C., Panayiotopoulos, T.: An attribute grammar interpreter as a knowledge engineering tool. Applied Informatics 9/86, (1986) 382-388
6. Clocksin, WF and. Mellish, C.S.: Programming in PROLOG
7. Panagopoulos, I., Pavlatos, C.and Papakonstantinou,G. :An Embedded System for Artificial Intelligence Applications, International Journal of Computational Intelligence, 2004
8. Panagopoulos, I., Pavlatos, C.and Papakonstantinou, G.: An Embedded Microprocessor for Intelligence Control, Journal of Rob. and Intel. Systems
9. Fu, K.: Syntactic Pattern recognition and Applications, Prentice-Hall 1982
10. Chen, H., Chen, X.: Shape recognition using VLSI Architecture, The International Journal of Pattern Recognition and Artificial Intelligence, 1993
11. Aho, A., Sethi, R., and. Ullman, J.: Compilers – Principles, Techniques and Tools. Reading, MA:Addison-Wesley, 1986, pp. 293-296
12. Demers, A., Reps, T., and Teitelbaum, T.: Incremental evaluation for attribute grammars with application to syntax-directed editors, in Conf. Rec. 8th Annu. ACM symp. Principles Programming Languages, Jan.1981,pp.415-418
13. Pavlatos, C. , Panagopoulos, I. , Papakonstantinou, G,: A programmable Pipelined Coprocessor for Parsing Applications, Workshop on Application Specific Processors (WASP) CODES, Stockholm, Sept. 2004
14. Chiang, Y., Fu, K.: Parallel parsing algorithms and VLSI implementation for syntactic pattern recognition". IEEE Trans. on Pattern Analysis and Machine Intelligence, PAMI-6 (1984)
15. Pavlatos C., Dimopoulos A. and Papakonstantinou G.: An Intelligent Embedded System for Control Applications, Workshop on Modeling and Control of Complex Systems, Cyprus,2005
16. www.xilinx.com/products/design_resources/proc_central/grouping/picoblaze.htm
17. Floyd, R.: The Syntax of Programming Languages-A Survey. IEEE Transactions on Electr. Comp., Vol. EC 13, No 4, (1964)
18. Earley, J.: An efficient context–free parsing algorithm. Communications of the ACM, Vol.13, (1970) 94-102
19. Graham, S.L., Harrison, M.A., Ruzzo, W.L.: An Improved context – free Recognizer. ACM Trans. On Programming Languages and System, 2(3) (1980) 415-462