A Parallel Parsing VLSI Architecture for Arbitrary Context Free Grammars

Andreas Koulouris, Nectarios Koziris, Theodore Andronikos,
George Papakonstantinou, Panayotis Tsanakas

National Technical University of Athens
Department of Electrical and Computer Engineering
Computer Systems Laboratory
Zografou Campus, 15773, Athens, Greece

Abstract

In this paper we propose a fixed size one-dimensional VLSI architecture for the parallel parsing of arbitrary context free (CF) grammars, based on Earley's algorithm. The algorithm is transformed into an equivalent double nested loop with loop-carried dependencies. We first map the algorithm into a 1-D array with unbounded number of cells. The time complexity of this architecture is O(n), which is optimal. We next propose the partitioning into fixed number of off-the-shelf processing elements. Two alternative partitioning strategies are presented, considering restrictions, not only in the number of the cells, but also in the inner structure of each cell. In the most restricted case, the proposed architecture has time complexity $O(n^3/p*k)$, where p is the number of available cells and the elements inside each cell are at most k.

Index Terms --- *Parallel parsing, Earley's algorithm, partitioning, systolic array, mapping.*

1. INTRODUCTION

Context-free grammars combine both the expressive power and the simplicity in their analysis. They can describe many features of natural languages and are widely used in syntactic pattern recognition applications. Many efficient parsing algorithms have been developed for this specific class of grammars [1-7].

The most known algorithms in the literature are CYK and Earley, which are dynamic programming procedures. Both can be applied to a general CF grammar, but CYK requires the grammar to be transformed in Chomsky normal form. The time complexity of these algorithms is, in general, $O(n^3)$, where n is the length of the input string. This time complexity can be a significant overhead for a reasonably large n. Consequently the efficient parallelization of these algorithms is of particular importance to the above areas.

Most of the proposed parallel parsing algorithms are based on CYK algorithm, thanks to it's simpler form. The problem of mapping the CYK algorithm into a VLSI architecture with both unbounded and fixed number of processors has been considered by Cheng & Fu in [8] and Ibbara et al. in [9,10]. However the transformation of a general CF grammar to an equivalent grammar in Chomsky normal form, may drastically increase the size of the grammar. Therefore a parallel recognizer of arbitrary CF grammars, based on Earley's algorithm is of great importance. Chang & Fu proposed in [11] a 2-D array architecture for the parallel implementation of Earley's algorithm with unbounded number of processors. Recently Ra et al. proposed a parallel implementation of Earley's parsing algorithm into an array of processors [12]. However, their architecture operates asynchronously and communication is based on message passing. For Earley's algorithm, asynchronous architectures are less efficient than synchronous ones for two reasons: First, they present a regular and repetitive data communication pattern; thus startup message latency and transmission delays override the net computation time. Second, even a very large grammar can be encoded in a total of a few digital words using a bit-vector representation [11].

On the other hand, the feasibility of the automatic parallelization of some special classes of sequential algorithms like nested DO (FOR) – loops has been examined in [13-18]. The minimum execution parallel time, as well as, upper and lower bounds for the number of cells, needed to achieve that optimal time, were elaborated for this special case, providing with optimal methods [13, 14, 15,19].

In this paper, we propose a synchronous VLSI architecture for the implementation of Earley's parsing algorithm, based on a general method of parallelization of nested loops [13]. This method makes an efficient mapping of the loop iterations, using the less possible processing elements. First, Earley's algorithm is mapped into an one-dimensional array, consisted of O(*n*) processing elements. Each element has a number of main operators (which are

responsible for executing the steps of Earley's algorithm) instead of only one as in [11]. Specifically, the number of basic elements inside the cell is an upper limit of the length of the input string that can be recognized by the architecture. Since the main operations in Earley's algorithm can be implemented in hardware with a few registers and simple gates (XOR, OR, AND, e.t.a.) [6,11], this is a realistic approach. This architecture can recognize an input string of size n, in exactly n+1 time steps (thus (O(n)). We then propose a partitioning strategy into fixed number of cells, having O(n) operators inside each cell. We finally consider the most restricted partitioning case into a fixed number p of cells with a bounded number k of operators inside each cell, having $k*p \geq n$. In this case, we cascade several cells with fixed number of operators, in order to construct a virtual cell having the required number of operators. This architecture has O($n^3/p*k$) time complexity, which coincides to the previous partitioning case, O(n^2/p), if $k=n$, and to the mapping case, O(n), when $k=p=$O(n).
The rest of the paper is organized as follows: In Section 2, we briefly give the basic notations and definitions of the CFG and Earley's parsing algorithm. In Section 3, we transform the Earley's algorithm into an equivalent 2-D nested loop. In Section 4, the optimal mapping of the Earley's algorithm into an one-dimensional VLSI array is presented. Finally, in Section 5, we present two partitioning cases, supporting both bounded and unbounded number of components (operators) inside each cell, together with illustrative examples.

2. Basic Concepts

We briefly give the definitions and the notations, used throughout this paper.

Definition 1. A Context Free Grammar (CFG) is a quadruple G=(V, N, P, S), where:

- V is the set of the symbols of the grammar, N is the set of the non-terminal symbols (T=V-N is the set of the terminal symbols)
- $P \subseteq N \times V^*$ is the set of the rules of the grammar, which are of the form $A \rightarrow a$, where $A \in N$ and $a \in V^*$. We call the symbol A the left hand symbol of a rule (LHS) and the symbol(s) α the right hand (RHS).
- S is the start (non-terminal) symbol of the grammar.

Definition 2. Let '•' be a symbol not in V. Then a rule $A \rightarrow \alpha \bullet \beta$ ($A \rightarrow \alpha\beta$ is in P) is called "*dotted rule*" and means that the α part of the rule has been found consistent with the input string, while the β part still needs to be considered.

Definition 3. The set *PREDICT*(B), $B \subseteq N$ and *PREDECESSOR*(A), $A \subseteq N$ are defined as:
PREDICT(B) $=\{C \rightarrow \gamma \bullet \delta \mid C \rightarrow \gamma\delta$ is in P, $\gamma \Rightarrow^* \varepsilon$, $B \Rightarrow^* C\eta$ for some B in R and some $\eta\}$
PREDECESSOR(A) = $\{B \mid B \Rightarrow^* A, B \in N\}$, that is the set of all the symbols that generate A

Many versions of the initial Earley's algorithm can be found in the literature [1-9]. In this paper, the form presented by Fu is used. It constructs a parsing table Tnxn, whose elements t_{ij} are sets of dotted rules. A string is correctly recognized, if at the element t_{0n} there is a doted rule of the form $S \rightarrow a \bullet$. Formally the algorithm is given bellow [11].

Algorithm 1 (Earley's parsing algorithm)

```
FOR i=1 to n do
  $t_{j-1,j} = Y \otimes \{a_j\}$
FOR j=2 to n do
 begin
  FOR i=0 to j-2 do
   $t_{ij} = t_{ij} \otimes \{a_j\}$
  FOR k=j-1 down to 0 do
   FOR i=k-1 down to 0 do
    $t_{ij} = t_{ij} \cup t_{ik} \otimes t_{kj}$
end
IF there is a rule $S \rightarrow \alpha \bullet$ in $t_{0,n}$ accept the input string
```

In above algorithm Y=*PREDICT*(N) and the operator ⊗ is defined as follows (λ symbolize the null string):

- Let Q be a set of dotted rules and $R \subseteq T$
$$Q \otimes R = \{A \rightarrow \alpha U \beta \bullet \gamma \mid A \rightarrow \alpha \bullet U \beta \gamma \in Q,\ \beta \Rightarrow^* \lambda,\ U \in R\}$$
and
$$\{B \rightarrow \delta C \xi \bullet \eta \mid \gamma = \lambda, B \rightarrow \delta \bullet C \xi \eta \in Y, \text{and } \xi \Rightarrow^* \lambda, C \Rightarrow^* A\}$$
- Let Q,R be a set of dotted rules
$$Q \otimes R = \{A \rightarrow \alpha U \beta \bullet \gamma \mid A \rightarrow \alpha \bullet U \beta \gamma \in Q,\ B \Rightarrow^* \lambda,\ U \rightarrow \delta \bullet \in R\}$$
and
$$\{B \rightarrow \delta C \xi \bullet \eta \mid \gamma = \lambda, B \rightarrow \delta \bullet C \xi \eta \in Y, \text{and } \xi \Rightarrow^* \lambda, C \Rightarrow^* A\}$$

Finally, the terms processing element cell and processor will be interchangeably used.

3. Data Representation

The main characteristics of a VLSI array are, first, its synchronous and regular flow of constant length data among neighboring cells, and second, the same simple and regular internal structure of each cell. In order to fulfill these requirements, we should represent the array elements t_{ij} in a compact form. In addition to this, the internal ⊗ operator should be as simple as possible.
By the definition of the ⊗ operator, one can see that it operates on a set of rules. Specifically the operation Q⊗R is divided into the following steps:
1. The set of all the left symbols of the rules in R (in which all the right part has been read) is calculated.
2. All the rules in set Q, which contain any element of the above set at the right of the dot, are found. In these rules, the dot is moved one place to the right.
3. All the rules in set Q, in which all the right part has

been read, are found. The corresponding set of the predecessors of their left-hand symbol is computed.
4. Finally, all the rules in the set PREDICT(N) are computed in the same way as in step 2.
The result is the union of the sets of the rules found in steps 2 and 4. A similar procedure is executed when R is a set of symbols and not a set of rules.
In general, the above steps have different execution times, depending on the index of the elements Q, R. This problem is solved if we represent the grammar with bit-vectors as it was proposed by Chang & Fu in [11].
In the following, a formal description of the implementation of the operator ⊗ in an PASCAL-like algorithm is given, which can be used in a preprocessing level as input to an automatic hardware synthesis tool. Moreover, the proposed implementation allows the grammar to support also ε-productions. (In [11] only ε-free grammars are considered as input). The removing of the ε-productions may duplicate the rules of the grammar, thus duplicating the length of the data that travel through the VLSI array. Since, in our implementation, ε-productions are also allowed, the overall execution time is significantly reduced.
The implementation of the operator ⊗ is illustrated as follows: If the grammar has *s* symbols, each symbol is encoded into a non-zero *s*-bit vector. In this encoding every *s*-bit vector differs only in one bit from the others. The set of the predecessors for each non-terminal is similarly encoded into a *s*-bit vector. The value of this bit - vector is derived by or-ing all the bit-vectors, that represent the symbols, which belong to this set. For each non-terminal symbol we use its encoding and the encoding of the set of its predecessors. If the grammar has *n* non-terminal symbols, we store at each cell an array containing *2n s*-bit vectors. We use the notation *symb[i],* to point the i-th symbol in this 2n array. Each rule of the grammar is encoded as an array of *s*-bit vectors. The first bit-vector is the LHS of the rule and the others the RHS. If the grammar has *w* rules and each rule has at most *r* symbols at the right part, then the rules of the grammar are stored in a *w*(*r*+1) array of k bit–vectors. We use the notation *rule[i],* to point the position of the i-th rule (i.e. its LHS). Finally we store at each cell three arrays of *w* (*r*+1)bit-vectors. Each row in these matrices represents a rule (only the RHS part), and each bit in the (*r*+1)-bit vector a position of the dot. The first matrix contains the set Y=PREDICT(N). In the second matrix (called M) the bit which denotes the end of the RHS of the corresponding rule is marked in each bit-vector. The third matrix, symbolized as E, is called the *‘empty’* matrix. In each row of E, we mark the bit(s), which correspond to symbols that produce the empty string. This matrix is used to transform dynamically the grammar into an equivalent ε-free form, without increasing the size of the main cell. Finally the elements of the parsing matrix T have the same form as the matrices Y, E, M (i.e. a *w*(*r*+1) array of bits).

The algorithm is executed in a 5-step procedure. It takes as input the sets Q, R in the form of a W(*r*+1) bit arrays and calculates the set (matrix) T = Q⊗R. We use the internal variables U, PRED of the form *s*-bit vector and B of *r*+1 bit-vector with initial value 0. Finally with A (a *s*-bit vector) we symbolize the representation of a symbol of the input string. In the following, algorithm the operator $\underline{\triangle}$ symbolizes the vectorial AND where:

$$\text{vectorial AND:} \quad \begin{cases} 1 \text{ if } (\, x \,\&\, y \, \neq 0) \\ 0 \ \text{ otherwise} \end{cases}$$

Algorithm 2 (operator ⊗)

```
FOR i=1 TO p        /* step1 */
   U=U OR rule[i]*(R[i] △ M[i])
U=U OR A
FOR i=1 TO w        /* step2 */
 begin
  B=Q[i]
  FOR j=1 to r
    B[j]=B[j] * (rule[i+j] △ U)
  T[i]=B>>1;
  FOR  j=2 to r+1
   T[i][j]=T[i][j]OR T[i][j-1]*E[i][j]
end
U=0
FOR i=1 to w       /* step 3 */
 U=U OR rule[i]*(T [i] △ M[i])
FOR i=1 to n       /* step 4 */
  PRED=PRED OR symb[i+1]*(symb[i] △ U)
FOR i=1 TO w      /*  step 5 */
 begin
   T[i][2]=T[i][2] OR (rule[i+1]△PRED)
   FOR  j=2 to r
    T[i][j]=T[i][j] OR T[i][j-1]*E[i][j]
end
```

Notice that with the above representation a null bit-vector represents the ‘neutral’ element for the operator ⊗. This gives us the possibility to preserve the regular data flow even if, at some time steps we transmit null values. All bit operations can be implemented in parallel, implying even the lowest level inherent parallelism of the algorithm.

4. DEPENDENCE ANALYSIS

The algorithm 1 can be rewritten in an equivalent form of perfectly double nested loop:

Algorithm 3 (Modified Earley’s algorithm)

```
FOR i=1 TO n
 FOR j=i-1 TO 0
  $t(i,j)=t(i-1,j) \otimes a_i \cup t(i-1,j) \otimes t(i,i-1) \otimes \cup\ldots\cup t(j+1,j) \otimes t(i,j+1)$
  END
END
```

It can be easily seen that the above algorithm calculates the recognition matrix row by row, so that the last element that will be computed is t(n,0) instead of t(0,n).
For the efficient parallelization of the algorithm 3, we use the method proposed by Andronikos et al. in [13]. This method partitions the index space of the above double nested loop into the less possible uniform disjoint chains of computations. After the partitioning phase, it assigns each chain to a different processing element, while preserving the dependence relations between the loop iterations. The resulting schedule is proved to be optimal both in terms of time and processor utilization.
In our algorithm the set of all possible dependence vectors (called Dependence Set, and symbolized DS), which relate different loop iterations is:

$$DS=\{d_i(i-k,0), d_j(0,j-k) \mid 1\leq i-k \leq n, 1-n \leq j-k \leq 0)$$

From the above set it is clear that satisfying dependencies $d_1(1,0)$ and $d_2(0,-1)$ all other dependencies will automatically be satisfied. This is illustrated in figure 1 where the index space and the dependence vectors are presented (for n=6).

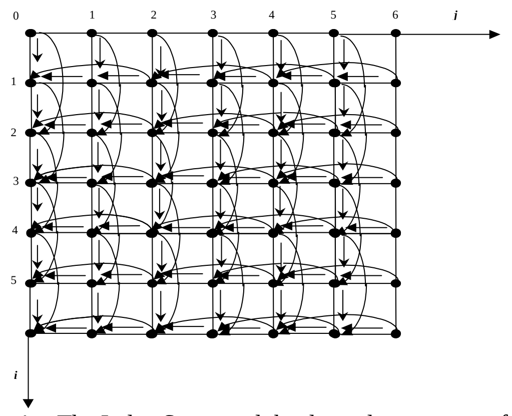

Figure 1: - The Index Space and the dependence vectors for algorithm (For simplicity reasons, only the first two dependencies (i.e. d (k, 0)) and d(0,-k), k=1,2) are shown).

5. MAPPING ONTO AN 1-D VLSI ARRAY

Unlike the empirical approach by Fu in [11], a systematic way, for mapping algorithm 3 to hardware, is used. More specifically, we applied the method [13], which leads to optimal time and space schedules. Table 1 presents, the time-schedule for n=6. Processor P_i is responsible for the computation of all the elements in the i-th column of the recognition matrix t
It is clear that, by applying the above mapping, the proper data flow is ensured.
For example, the computation of element t(4,0) will be done by processor p_1 at the 4-th time step. From algorithm 3 we see, that for the computation of the element t(4,0), the values of the elements t(4, k), t(k, 0) ($j \leq k \leq i-1$) are needed. By this time, all these elements (i.e. T(1,0), t(2,0), t(3,0), t(4,1), t(4,2), and t(4,3)) have already been computed and sent to processor p_1.

	Processing Elements					
Time	P_1	P_2	P_3	P_4	P_5	P_6
1	t(1, 0)	t(2, 1)	t(3, 2)	t(4, 3)	t(5, 4)	t(6, 5)
2	t(2, 0)	t(3, 1)	t(4, 2)	t(5, 3)	t(6,4)	-
3	t(3, 0)	t(4, 1)	t(5, 2)	t(6,3)	-	-
4	t(4, 0)	t(5, 1)	t(6,2)	-	-	-
5	t(5, 0)	t(6,1)	-	-	-	-
6	t(6,0)	-	-			

Table 1: -. Time schedule for n=6. Each column of the matrix T is assigned to a different cell

Assigning the computation of each column of the recognition matrix to the same processor, leads to one-dimensional vlsi architecture of n processors. This architecture is illustrated in figure 2. Each link is used to transfer both the element t(i,j) and the input symbol a_i, within one time step (recall from section 3 that both are represented as bit-vectors).

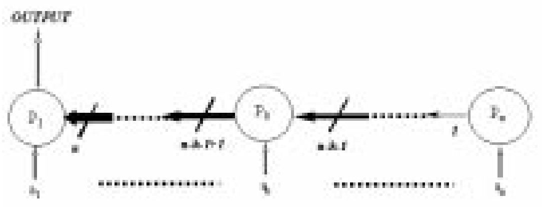

Figure 2: - One dimensional architecture for the parallel implementation of Earley's algorithm

The input string $S=a_1a_2...a_n$ is initially loaded, in parallel, to processing elements, so that P_i processor has a_i character (Figure 2).
The number of operators implemented inside each cell is at most equal to the length of the input string as it is shown in figure 3. Most specifically, P_i processor contains i operators $\otimes$.
With this architecture, we can obtain the result from the processor 1 at exactly n time steps.
The VLSI cell is illustrated in Fig 3. Since we represent the set of rules with an array of bit vectors, it is obvious that the operator $\cup$ can be implemented by OR gates. Since each cell computes the corresponding t(i,j) within one time instance, we use an additional 1D delay to synchronize the transfer of data from input to output. Thus, the data from *Input$_i$* are forwarded to *Output$_i$*

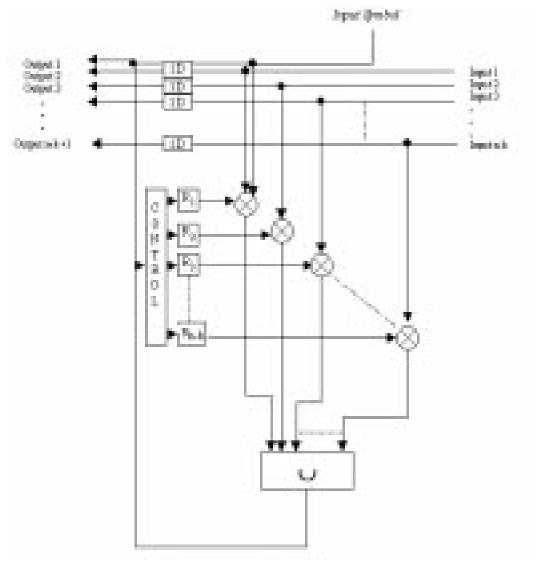

Figure 3 :- The internal structure of the cell P_k.

synchronously with the result from the $\cup$ operator.

The P_k cell has n-k inputs and n-k+1 outputs, so that neighboring cells have the same number of interconnection links (Figure 2).

Finally there are n-k $\geq$ 1 registers which are driven by a control unit. The registers are loaded one by one each time step. Specifically, the output of the $\cup$ operator e is loaded to R_i at the i-th time instance.

Each cell performs the following operation:

$$\textit{Cell output} = \textit{operation } 1 \cup \ldots \cup \textit{operation } n\text{-}k$$

$$\text{where: } \textit{operation } i = R_i \otimes \textit{Input}_i$$

The above interconnection strategy, implements the following relations:

$$\text{Output}_1 = \textit{Cell output}$$

$$\text{Output}_i = \textit{Input}_{i-1} \text{ for } 2 \leq i \leq n\text{-} k+1$$

In table 2, the operation of the 4-th cell, for an input string of maximum length n=7, is summarized.

The proposed architecture has the following advantages over the so far presented implementations: First, it has a single time clocking and simple data flow. Second, it only needs one copy of the encoded grammar (symbols, rules, predecessors etc) inside each processing element. Finally, on the contrary with the 2-D architecture of [11], we have at the i-th time step, the result of the parsing of the substring $S^i = a_1a_2 \ldots a_i$, as the output of the P_1 cell. This could be very useful in cases where, we want to collect information about a sub-string of the entire string (e.g. in some pattern recognition problems).

6. ALGORITHM PARTITIONING INTO FIXED SIZE OF CELLS

For the majority of the applications the maximum length is small, fixed and known beforehand (e.g. syntactic recognition of the ECG signal). In the above Section, we assumed unbound number of cells, equal to the maximum length of the input string. This means, that the internal structure of each cell depends on the maximum length. However, it would be ideal to produce a general-purpose cell with a fixed internal structure. If we need to implement an array for a specific application (thus an arbitrary fixed maximum length) wee only connect several such kind of cells. In addition to this, we may also have limitations on the number of off-the-shelf cells. In this section we present a partitioning strategy, allowing for bounded number of cells and fixed number of internal operators. We follow the LPGS (Locally Parallel Globally Sequential) approach, which is widely used in systolic array partitioning [19]. The virtual array of cell is divided into blocks, whose size is equal to the number of the available cells. Inside every block each virtual processor is assigned to the corresponding physical one. Once the block is completed, the same physical array of cells is used to implement the next block and so on. We first present a partitioning scheme, where we have bounded cells but the

Time	Input Symbol	R_1	R_2	R_3	lnput 1	Input 2	Input 3	Operation1	Operation2	Operation3	Out1
Initial	a_4	0	0	0	0	0	0	-	-	-	-
1	- 0	- t(4,3)	- 0	- 0	- t(5,4),a_5	- 0	- 0	Y$\otimes a_4$ -	0 -	0 -	t(4,3) -
2	- 0	- t(4,3)	- t(5,3)	- 0	- t(6,4)	- t(6,5),a_6	- 0	t(4,3) $\otimes$ t(5,4) ,a_5 -	0 -	0 -	t(5,3) -
3	- 0	- t(4,3)	- t(5,3)	- t(6,3)	- t(7,4)	- t(7,5)	- t(7,6),a_7	t(4,3) $\otimes$ t(6,4) -	t(5,3) $\otimes$ t(6,5) ,a_6 -	0 -	t(6,3) -
4	- 0	- t(4,3)	- t(5,3)	- t(6,3)	- 0	- 0	- 0	t(4,3) $\otimes$ t(7,4) -	t(5,3) $\otimes$ t(7,5) -	t(5,3) $\otimes$ t(6,5),a_7 -	t(7,3) -

Table 2. The time scheduling inside each cell. During a clock cycle; first the values at the registers and at the input lines are used to be computed the next element t(i,3) and next the output is stored to the corresponding register

number of the internal operators depends on the maximum length.

a. Fixed number of processors and unbounded number of ⊗ operators

Let us symbolize with *p* the number of processors, *k* the number of operators inside each cell, and with *n* the length of the input string. In this case it is assumed that k ≥ n-1.
Recall from table 1, that the i-th processor calculates the i-th column in exactly n-i+1 time steps. We divide the total of n-columns in n/p blocks of p columns each. Each block of p columns is mapped to the available set of p cells, following the mapping strategy presented in Section 5. The j-th block will complete its computations at (j-1)*p time steps. Adding all the computations for the entire length of string we have:

$$\text{Total Time} = p + 2 * p + 3 * p + ... + n * p =$$

$$p * \frac{\frac{n}{p} * (\frac{n}{p} + 1)}{2} = O(n^2 / p) \qquad (1)$$

The architecture is a ring of p processors with additional memory modules and delays as feedback of the results of the previous block. In figure 4 we show the block diagram for p=3 and n=6.
In this partitioning, the data that come out of P_1 are automatically arranged using only some register buffers and delayers. Specifically each output line of the last processor P_1 is stored to a register (filled in the same way as that of the cell). A p-time step delay is added before the output of the register is forwarded to the corresponding input of the processor P_p. This procedure is graphically shown in figure 5.

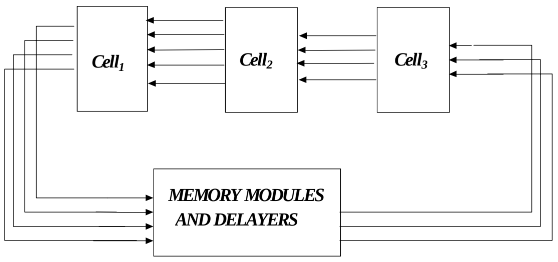

Figure 4: - The ring architecture for n=6, p=3.

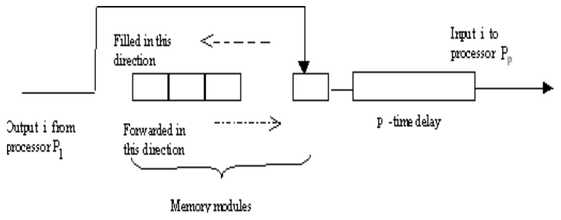

Figure 5: - The connections inside the Control Unit for the automatic re-arrangement of the data

Once a block finishes its calculations, we reset the registers of the cells to prepare for the calculation of the next block.
In tables 3.i the detailed data flow inside the partitioned architecture is shown. Notice that the P_i processor calculates columns i+j*p where, 0≤ j ≤ n/p-1. In this example 0≤ j ≤ 1 and the result is obtained after 9 time steps instead of 6 (relation (1)).

b. Fixed number of processors and ⊗ operators inside each cell

Table 3.1

Time	Out1	Out2	Out3	Out4	In1	In2	In3
1	t(4,3),a_3	-	-	-	-	-	-
2	t(5,3)	t(5,4) ,a_5	-	-	-	-	-
3	t(6,3)	t(6,4)	t(6,5),a_6	-	-	-	-
4	t(1,0),a_1	-	-	-	-	-	-
5	t(2,0)	t(2,1),a_2	-	-	t(4,3), a_4	-	-
6	t(3,0)	t(3,1)	t(3,2),a_3	-	t(5,3)	t(5,4), a_5	-
7	t(4,0)	t(4,1)	t(4,2)	t(4,3),a_4	t(6,3)	t(6,4)	t(6,5),a_6

* Out i = the output i from processor P_1 , In i = the input i to processor P_3

Table 3.3

Time	Input String	R_1	R_2	R_3	R_4	Input1	Input2	Input3	Input4	Out
1	a_5	-	-	-	-	-	-	-	-	t(5,4)
2	-	t(5,4)	-	-	-	t(6,5)	-	-	-	**t(6,4)**
3	-	t(5,4)	t(6,4)	-	-	-	-	-	-	-
4	a_2	*reset*	*reset*	*reset*	*reset*	-	-	-	-	t(2,1)
5	-	t(2,1)	-	-	-	t(3,2)	-	-	-	t(3,1)
6	-	t(2,1)	t(3,1)	-	-	t(4,2)	t(4,3), a_4	-	-	t(4,1)
7	-	t(2,1)	t(3,1)	t(4,1)	-	t(5,2)	t(5,3)	t(5,4), a_5	-	t(5,1)
8	-	t(2,1)	t(3,1)	t(4,1)	t(5,1)	t(6,2)	t(6,3)	t64	t(6,5)	**t(6,1)**

Table 3.2

Time	Input String	R_1	R_2	R_3	Input 1	Input 2	Input 3	Out
1	a_6	-	-	-	-	-	-	**t(6,5)**
2	-	t(6,5)	-	-	-	-	-	-
3	-	t(6,5)	-	-	-	-	-	-
4	a_3	*reset*	*reset*	*reset*	-	-	-	t(3,2)
5	-	t(3,2)	-	-	t(4,3), a_4	-	-	t(4,2)
6	-	t(3,2)	t(4,2)	-	t(5,3)	t(5,4), a_5	-	t(5,2)
7	-	t(3,2)	t(4,2)	t(5,2)	t(6,3)	t(6,4)	t(6,5), a_6	**t(6,2)**

Table 3.4

Time	Input String	R_1	R_2	R_3	R_4	R_5	Input 1	Input 2	Input 3	Input 4	Input 5	Out
1	a_4	-	-	-	-	-	-	-	-	-	-	t(4,3)
2	-	t(4,3)	-	-	-	-	t(5,4), a_5	-	-	-	-	t(5,3)
3	-	t(4,3)	t(5,3)	-	-	-	t(6,4)	t(6,5), a_6	-	-	-	**t(6,3)**
4	a_1	*reset*	*reset*	*reset*	*reset*	*reset*	-	-	-	-	-	t(1,0)
5	-	t(1,0)	-	-	-	-	t(2,1)	-	-	-	-	t(2,0)
6	-	t(1,0)	t(2,0)	-	-	-	t(3,1)	t(3,2), a_3	-	-	-	t(3,1)
7	-	t(1,0)	t(2,0)	t(3,0)	-	-	t(4,1)	t(4,2)	t(4,3), a_4	-	-	t(4,0)
8	-	t(1,0)	t(2,0)	t(3,0)	t(4,0)	-	t(5,1)	t(5,2)	t(5,3)	t(5,4),a_5	-	t(5,0)
9	-	t(1,0)	t(2,0)	t(3,0)	t(4,0)	t(5,0)	t(6,1)	t(6,2)	t(6,3)	t(6,4)	t(6,5),a_6	**t(6,0)**

An example of the operation of the ring architecture for parsing an input string(n=6) having only 3 processing elements. **(Table3.1:** The data flow between the ring architecture and the control unit,**Table 3.2:** Time scheduling in 3-rd processing element,**Table 3.3:** Time scheduling in 2-nd processing element,**Table 3.4:** Time scheduling in 1-st processing element)

In this paragraph we will present a partitioning strategy using off-the-shelf processing elements. In other words, the internal structure of each cell is predefined. The designer has to use a fixed number of such cells to construct an array, which recognizes, up to a specific length, input strings. Specifically, the maximum length of the input string is determined by the number *p* of available cells, as well as by the number *k* of the operators inside each cell.

The key issue behind the proposed partitioning is the cascading of several cells to create a larger containing the required number of ⊗ operators. We thus construct a bigger, virtual cell. After this initial cascading, we apply the partitioning of the previous paragraph to implement the whole array. It is clear that this approach depends on the product *p***k*.

Thanks to the associative property of the ∪ operation, the right result can be obtained by oring the outputs of the cascaded cells. The cell's control unit is slightly modified as shown in figure 6.

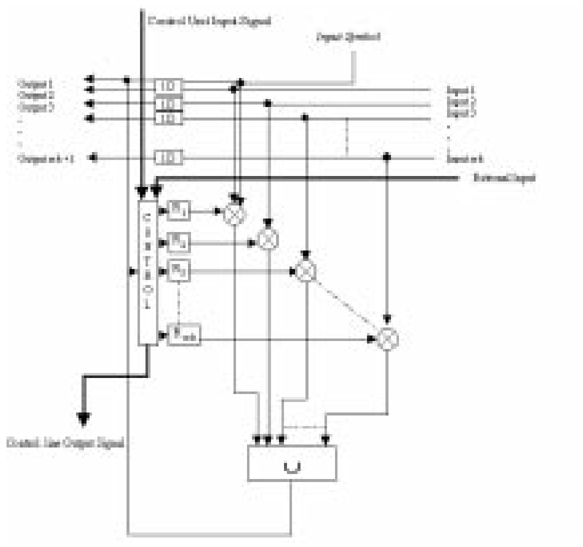

Figure 6:- The internal structure for a processing element, which can be connected with other similarly structured elements, in order to build a cell with larger number of operators ⊗

The following links/signals are added: the Control Unit Input Signal, the Control Unit Output Signal, and the External Input. The data from the External Input are copied to registers R_i (in the same way as the output of the ∪ operator) with an appropriate signal from the control unit of the cell. It is clear, that this enhanced cell can be also used without a modification in the mapping phase of Section 4. If the Control Input Signal is not activated, the cell works as in figure 3. This means, that the registers of the cell are filled by the output of the ∪ module. On the contrary, if the signal is activated, the cell is working in the "cascaded" mode, and the registers are filled by the external input.

Suppose, for example, that we want to cascade two cells of 3 basic elements, in order to build a virtual cell of 6 basic elements. The interconnection is shown in figure 7.

Initially, the control unit of $cell_1$ works, while the control unit of $cell_2$ is halted. When all the registers of $cell_1$ are filled, the control unit sends to the corresponding unit of $cell_2$ a signal. This signal forces the data in External Input line of $cell_2$ to be copied to register R_1 and also activates the Control unit of $cell_2$ in "cascade" mode. Thus, by the next time instance, the output of $cell_2|cell_1$ will be copied to the corresponding registers. Notice that the control unit of $cell_1$ will be now halted and the output will not be copied to any register.

Consider the virtual cell of figure 7 used to parse an input string of length 7. This virtual cell is consisted of two cascaded cells with 3 inputs and 4

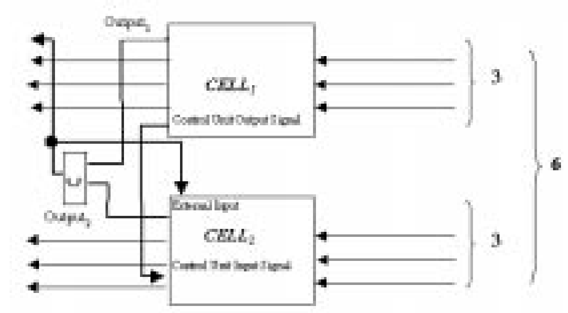

Figure 7:- Connection of two cells with 3 ⊗ operators, in order to build a virtual cell with 6 ⊗ operators

outputs, for a total of six inputs and seven outputs, thus responsible for the computation of the first column of the parsing table (i.e. the computation of the elements t(i,0) $1 \leq i \leq 7$). Obviously such cascaded is not necessary for all columns, since the required inputs and ⊗ operators are less than three for P_i $i \leq 4$.

Initially $cell_1$ will compute the element t(1,0) using the input symbol a_1 (from input line). This value will be stored in register R_1. As already seen in Section 4, after 3 time steps the registers of $cell_1$ will have the values: R_1=t(1,0), R_2=t(2,0), R_3=t(3,0). Since all the registers of $cell_1$ are filled, the control unit will sent the corresponding signal to the control unit of $sell_2$. The next time step $cell_1$ will compute the value: t(4,0)=t(1,0) ⊗ t(4,1) (from $Input_1$) ∪ t(2,0) ⊗ t(4,2) (from $Input_2$) ∪ t(3,0) ⊗ (t(4,3),a_4) (from $Input_3$). The value t(4,0) will be copied to register R_1 of $cell_2$.

The values of the internal registers and input and output links, are next as follows:

<u>$Cell_1$:</u>

$Input_1$=t(5,1), ***$Input_2$*** = t(5,2), ***$Input_3$*** =t(5,3),
R_1=t(1,0), ***R_2***=t(2,0), ***R_3***=t(3,0),

$Output_1$= t(1,0)$\otimes$t(5,1) $\cup$ t(2,0)$\otimes$t(5,2) $\cup$ t(3,0)$\otimes$ t(5,3).
$Output_2$ = t(5,1), ***$Output_3$***= t(5,2), ***$Output_4$***=t(5,3).

Cell$_2$:
$Input_1$= t(5,4), a_5, ***$Input_2$***=0, ***$Input_3$***=0
R_1=t(4,0), ***R_2***=t(5,0), ***R_3***=0
$Output_1$=t(4,0)$\otimes$(t(5,4),a_5), ***$Output_2$***= ***$Output_3$***= ***$Output_4$***=0

Cascaded Cell :
$Output_1$ =t(4,0)$\otimes$(t(5,4),a_5), t(1,0)$\otimes$t(5,1) $\cup$ t(2,0)$\otimes$t(5,2) $\cup$ t(3,0)$\otimes$t(5,3) =t(5,0)
$Output_2$=t(5,1),***$Output_3$***=t(5,2),***$Output_4$***=t(5,3)
$Output_5$=t(5,4), a_5, ***$Output_6$***=0

which are exactly the outputs of a cell with 6 basic elements
Generally, if we have a string of maximum length of n and also holds: n-1=a*k, and p/a=b $\geq$ 1, then we use *a* cells in order to construct *b* virtual cells of n-1 basic elements (operators). Similar, according the LPGS partitioning algorithm of the previous paragraph, we can parse the string in O(n^2/b) time steps. Given an example, if we have 10 cells of 5 basic elements and we want to parse an input string of length 21 then we cascade 4 cells and we recognize the string in O(21^2/2) time steps.

7. CONCLUSION

An optimal one-dimensional VLSI architecture for the parallel execution of Earley's algorithm was presented in this paper. This architecture was derived by a general method of mapping nested loops onto VLSI architectures.
Two alternative partitioning methods of the above algorithm were presented, considering limitations not only in the number of the processing elements but also in the structural complexity inside each cell.

8. REFERENCES

1 K. S. Fu Syntactic Pattern Recognition and Applications, *Prentice-Hall 1982*
2 D. H. Younger "Recognition and parsing of context-free languages in time n^3," *Information and Control,* vol. 10, pp. 189-208, 1967.
3 V. Acho, J. D. Ullman The theory of Parsing Translating and Compiling vol. I *Prentice Hall* Inc.
4 J. Earley "An efficient context-free parsing algorithm," *Commun. of the ACM,* vol. 13, pp. 94-102, 1970.
5 L. Valiant "General context free recognition in less than cubic time," *Journal of Computer and System Science,* vol. 10, pp. 308-315, 1975.
6 S. L. Graham, M. A. Harrison and W. L. Ruzzo "On line context-free languages recognition in less than cubic time," in Proc, *8th Annu. ACM Symp. Theory of Comput* , May 1976.
7 T. Kasami "An efficient recognition and syntax analysis algorithm for context free languages," Science Report AF CRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Mass. 1965.
8 H.D. Cheng, K.S. Fu "Algorithm Partition and Parallel Recognition of General CFG Languages Using Fixed-Size VlSI Architecture," *Pattern Recognition,* vol 19, no 5, pp 362-372, 1986.
9 J. H. Chang, O. H. Ibbara, M. A. Palis "Parallel Parsing on a One-Way array of Finite-State Machines," *IEEE Trans. Comput.,* vol 36, no 1, pp. 64-75, 1987.
10 O.H. Ibarra, T.C. Pong, S.M. Sohn "Parallel Recognition and Parsing on the hypercube,*" IEEE Trans.Comp.* 40, pp 764-770, 1991.
11 Y. T. Chiang and K. S. Fu "Parallel parsing algorithms and VLSI implementation for syntactic pattern recognition," *IEEE Trans. Pattern Anal. Mach. Intell.* PAMI-7, 1985.
12 D.Y.Ra, J.H. Kim, "A parallel parsing algorithm for arbitrary context-free grammars," *Information Processing Letters* 58, pp. 87-96, 1996.
13 T. Andronikos, N. Koziris, Z. Tsiatsoulis, G. Papakonstantinou, P. Tsanakas "Lower Time and Processor Bounds for Efficient Mapping od Dependence Algorithms into Systolic Arrays," *Journal of Parallel Algorithms and Applications,* vol 10, pp. 177-194, 1997.
14 T. Andronikos, N. Koziris, G. Papakonstantinou, P.Tsanakas "Optimal Scheduling for UET/UET-UCT Generalized n-Dimensional Grid Task Graphs",*Proceedings of 11th IEEE/ACM International Parallel Processing Symposium (IPPS 97), pp. 146-151,* Geneva, Switzerland.
15 E. Bampis, C. Delorme, J.C. Konig, "Optimal Schedules for d-D Grid Graphs with Communication Delays," *Symposium on Theoretical Aspects of Computer Science (STACS96*), Grenoble France 1996.
16 N. Koziris, G. Papakonstantinou, P. Tsanakas, "Automatic Loop Mapping and Partitioning into Systolic Architectures," *Proceedings of the 5th Panhellenic Conference on Informatics*, Dec. 1995, pp. 777-790, Athens.
17 P. Z. Lee, Z.M. Kedem "Mapping Nested Loop Algorithms into Multidimensional Systolic Arrays," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no.1, pp. 64-76, 1990.
18 W. Shang, J.A.B. Fortes "Time Optimal Linear Schedules for Algorithms with Uniform Dependencies," *IEEE Trans. Comput.*, vol. 40, no. 6, pp. 723-742, 1991.
19 A. Darte, "Regular Partitioning for fixed-size systolic arrays," INTEGRATION, The VLSI Journal, vol. 12, pp. 239-304, 1991.