An FPGA-based coprocessor for the parsing of context-free grammars

Cristian Ciressan[†], Eduardo Sanchez[‡], Martin Rajman[†] and Jean-Cédric Chappelier[†]
École Polytechnique Fédérale de Lausanne,
Computer Science Dept., 1015-Lausanne, Switzerland,
{Cristian.Ciressan, Eduardo.Sanchez, Martin.Rajman, Jean-Cedric.Chappelier}@epfl.ch

Abstract

This paper presents an FPGA-based implementation of a co-processing unit able to parse context-free grammars of real-life sizes. The application fields of such a parser range from programming languages syntactic analysis to very demanding Natural Language Applications where parsing speed is an important issue.

1 Introduction

Formal languages, in particular those described by context-free (CF) grammars [11], are used in many applications such as syntactic pattern recognition [10], syntactic analysis of programming languages [1], natural language processing (NLP), etc. The parser speed for such formal languages is an important issue in the case of real-life applications in the above mentioned domains. For instance, in the particular case of industrial NLP applications, real-time constraints often have to be considered and efficient parsing solutions need to be proposed. Typical examples of such applications are:

- data production and processing : optical character recognition and spell checking, as well as advanced information retrieval [13, 8] and text mining [6, 7] techniques may all require parsing to further enhance performance by integrating syntactic knowledge about the language. When huge amounts of data need to be processed, efficient low complexity parsers are required;
- human-machine interface : state-of-the-art vocal interfaces use standard Hidden Markov Models (HMM) that only integrate very limited syntactic knowledge. Better integration of syntactic processing within speech-recognition systems is therefore an important goal; e.g., in the case where the output of the speech recognizer is further processed with a syntactic parser to filter out those of the hypotheses (i.e. sentences) that are not syntactically correct. The case of a sequential coupling, presented in [5], is such an example. Due to the real-time constraints in such an application, fast parsing is again required.

Various low complexity (i.e. polynomial) parsing solutions have been proposed for context-free languages, in particular several parallel implementations of the standard $O(n^3)$ time complexity[1] Cocke-Younger-Kasami (CYK) algorithm [1]. This algorithm is known to have $O(n^2)$ time complexity when executed on a 1D-array of processors and $O(n)$ time complexity when executed on a 2D-array of processors [9].

For such arrays, various VLSI designs have been proposed : a syntactic recognizer based on the CYK algorithm on a 2D-array of processors [4] and a robust (error correcting) recognizer and analyzer (with parse tree extraction) based on the Earley algorithm on a 2D-array of processors [3]. Although these designs meet the usual VLSI requirements (constant-time processor operations, regular communication geometry, uniform data movement), the hardware resources they require do not allow them to accommodate real-life CF grammars used in large-scale NLP applications[2].

In this context, we propose an FPGA-based 1D-array of processors implementation of the CYK algorithm able to accommodate real-life CF grammars that can parse input sentences of parametrical, i.e. customizable, maximal length. The design was described in VHDL, simulated for validation, synthesized and tested on an existing FPGA board, and finally compared for performance against two software

[†]Artificial Intelligence Laboratory; this work is funded by the FNRS grant # 21-52689.97
[‡]Logical Systems Laboratory

[1]Where n is the length of the input sentence
[2]For instance the CF grammar extracted from the SUSANNE corpus [12] contains more than 10,000 non-terminals and 70,000 grammar rules when written in Chomsky normal form

implementations. Its main features are:

- word-lattice parsing : the output of a speech recognizer is a number of possible sentences, often represented in the compact form of a word-lattice. Unlike the usual parsing algorithms that process sentences, our system is able to parse whole word-lattices and is therefore better adapted for integration in the framework of a speech recognition system. When integrating the parser within a speech recognition system, the ability to parse word-lattices is an important functionality that is required by a large number of applications relying on speech recognition interfaces;

- scalability : the system can be exactly tailored to the characteristics of any given Chomsky normal form (CNF) CF grammar so that no hardware resources are wasted;

- extensibility : the number of processors is not limited by the resources available in the FPGA. It can be increased on demand by cascading several FPGA circuits.

The design was described in VHDL in order to have a technology independent implementation that can be later used to target an ASIC implementation.

In section 2 we briefly present the CYK algorithm and the changes required for its adaptation to word-lattice parsing. Section 3 describes the design, its main components, and scalability and extensibility properties. Section 4 analyzes the performance of the FPGA design in comparison with two software implementations of the CYK algorithm. Conclusions and future extensions are presented in section 5.

2 CYK algorithm for word-lattice parsing

2.1 The CYK algorithm

A CF grammar is a 4-tuple $G = \{N, \Sigma, S, P\}$ where:

- N is a set of non-terminals (representing grammatical categories, e.g. verb V, noun-phrase NP, sentence S, ...);

- Σ is a set of terminals (representing words);

- $S \in N$ is the top level non-terminal (corresponding to a sentence);

- P is a set of grammar rules, i.e. a subset of $N \times (N \bigcup \Sigma)^*$ written in the form of $X \rightarrow \alpha$, where $X \in N$ and $\alpha \in (N \bigcup \Sigma)^*$. For instance, a grammar rule can be $S \rightarrow NP\,V$, representing that a sentence can consist of a noun-phrase followed by a verb;

We use capital letters X, Y, Z, ... to denote non-terminal symbols and a, b, ... to denote terminal symbols. The CYK algorithm is constraint to use CF grammars written in CNF : every grammar rule is either of the form $X \rightarrow YZ$ or $X \rightarrow a$. However, since any CF grammar can be rewritten in CNF, this constraint is not a theoretical limitation, but a practical one.

Assume that we have the CNF CF grammar $G = \{N, \Sigma, S, P\}$ and an input sentence $w_1 w_2 \dots w_n$, $n \geq 1$, and $w_i \in \Sigma$ is the i^{th} word in the sentence. Let $w_{ij} = w_i w_{i+1} \dots w_{i+j-1}$ be the part of the input sentence that starts at w_i and contains the next $j-1$ words and $N_{i,j}$ the subsets of N defined by $N_{i,j} = \{X \in N : X \Rightarrow^\star w_{ij}\}$, where $X \Rightarrow^\star w_{ij}$ means that w_{ij} can be derived from X by applying a succession of grammar rules.

Every set $N_{i,j}$ can be associated with the entry on column i and row j of a triangular parse table (see figure 1(a)) – henceforth referred as CYK table.

The CYK algorithm is defined as follows:

1: **for** $i = 1$ to n **do**
2: $\quad N_{i,1} = \{X : (X \rightarrow w_i) \in P\}$
3: $\quad$ **for** $j = 2$ to $n - i + 1$ **do**
4: $\quad\quad N_{i,j} = \emptyset$
5: $\quad$ **end for**
6: **end for**
7: **for** $j = 2$ to n **do**
8: $\quad$ **for** $i = 1$ to $n - j + 1$ **do**
9: $\quad\quad$ **for** $k = 1$ to $j - 1$ **do**
10: $\quad\quad\quad N_{i,j} = N_{i,j} \bigcup \{X : (X \rightarrow YZ) \in P$ with $Y \in N_{i,k}$ and $Z \in N_{i+k,j-k}\}$
11: $\quad\quad$ **end for**
12: $\quad$ **end for**
13: **end for**

The lines $1-6$ in the algorithm correspond to the initialization step, when the sets $N_{i,1}$ are initialized by only using the grammar rules of the form $X \rightarrow w_i$. In the CYK table this corresponds to the initialization of all entries on the bottom row. The lines $7-13$ correspond to the subsequent filling-up of the CYK table once the initial sets $N_{i,j}$ were constructed. Finally, the parsing trees can be extracted from the CYK table if necessary. If S is the topmost symbol (root) of

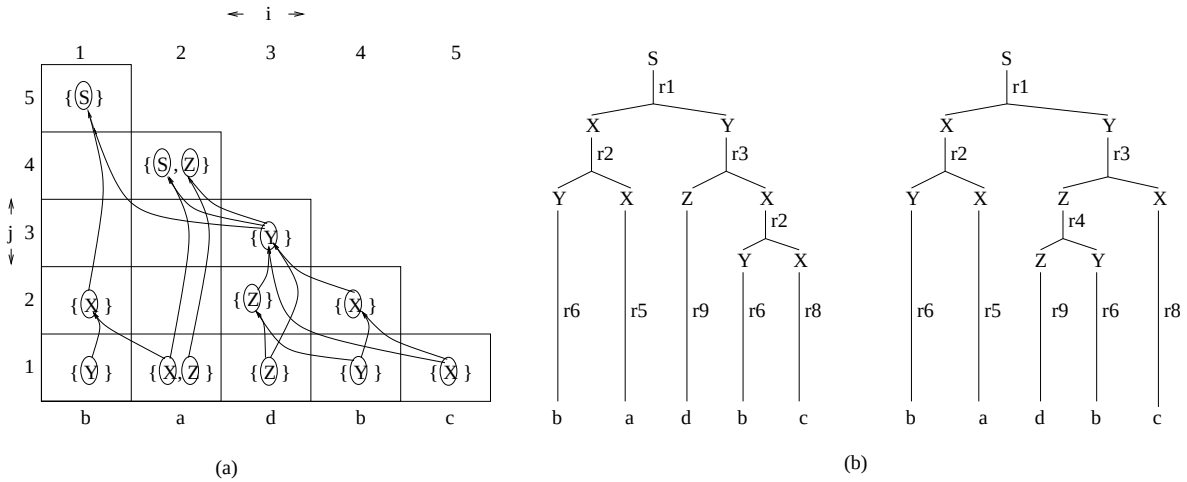

Figure 1: (a) CYK table initialization (j=1) and filling (j=2,3,4 and 5), (b) two possible parsing trees corresponding to the input sentence "`b a d b c`"

such a tree, the sentence is syntactically correct for the grammar G.

2.2 Example

Assume that the CF grammar G is given by:

$$N = \{S, X, Y, Z\},$$
$$\Sigma = \{a, b, c, d\},$$
$$P = \{S \to XY\ (r1),\ X \to YX\ (r2),\ Y \to ZX\ (r3),$$
$$Z \to ZY\ (r4),\ X \to a\ (r5),\ Y \to b\ (r6),$$
$$Z \to a\ (r7),\ X \to c\ (r8),\ Z \to d\ (r9)\}$$

where S is the top level non-terminal and $r1, \ldots, r9$, are used to denote the grammar rules. With this grammar, we want to parse the sentence "`b a d b c`". During the initialization step, only the rules $r5$ to $r9$ are used (see figure 1(a)) to fill-in the entries on the bottom line of the CYK table. During CYK table filling, the rows $j = 2, \ldots, 5$ are filled successively in this order, making use of the grammar rules $r1$ to $r4$. Finally, two possible parsing trees are extracted from the CYK table (see figure 1(b)).

2.3 Word-lattice adaptation

An example of word-lattice representation is given in figure 2. Each path starting in the leftmost node and ending in the rightmost node of the word-lattice corresponds to a possible recognized sentence. These sentences are subject to be filtered-out by the syntactic parser.

When dealing with word-lattices, the difference with the previously presented CYK algorithm is that

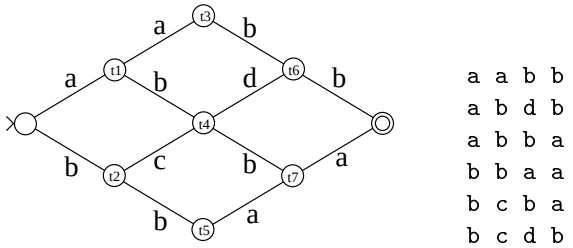

Figure 2: A toy word-lattice containing 6 sentences.

not only the sets $N_{i,1}$ may be initialized during the initialization step, but any of the sets $N_{i,j}$ as with word-lattice representation words may occur anywhere in the CYK table (see figure 3(b)). Thus, in order to adapt the CYK algorithm to word-lattice parsing, the initialization step needs to be extended [5].

To illustrate this point, let us consider the word-lattice given in figure 2, containing the six sentences "`a a b b`", "`a b d b`", "`a b b a`", "`b b a a`", "`b c b a`" and "`b c d b`" and the same grammar G as used in the former example. First the lattice nodes are ordered by increasing depth[3] (see figure 3 (a)). Such an ordering is natural in the case of lattices produced by a speech recognizer, in which nodes correspond to (chronologically ordered) time instants $t1$, $t2$, ...

This new representation of the word-lattice can be mapped over the CYK table as follows:

1. the intervals between successive nodes are associ-

[3] A node depth is the minimal number of arcs from the initial node, with random choice in case of equalities

ated to the column indices of the parsing table;

2. if the lattice arc (tm, tn) $(n > m)$ is labeled w, then the set $N_{m+1,n-m}$ corresponding to the CYK table entry $(m+1, n-m)$ is initialized with $\{X : (X \rightarrow w) \in P\}$ (see figure 3 (b)).

The hardware design we are going to present implements the CYK algorithm adapted for word-lattice parsing.

3 The FPGA Design

The general idea of our current hardware design is to use $n-1$ processors to parse sentences of at maximum n words. For example, the block diagram in figure 4 represents a 10 processor system that can parse any sentence of length less or equal to 11 words and that we have effectively implemented on a RC1000-PP[4] FPGA board. In the figure, the elements inside the dashed line are implemented within the on-board FPGA chip. The other elements (CYK and grammar memories) are implemented in SRAM chips also present on the board. Before any parsing can start, the grammar memories have to be configured with the binary image of the data-structure representing the CNF CF grammar (see section 3.4). Similarly, the CYK memory need to be initialized, where needed, with the structures used to represent the sets $N_{i,j}$ (see section 3.2) and the **GLOBAL controller** (**G-CTRL**) has to be initialized with the length of the sentence to be parsed. All initializations are done off-line in the current implementation. The **startPARSE** signal starts the parsing and the **overPARSE** signal indicates the end of the parsing. The parse result is available at some output **outPARSE** (not represented in figure 4) of each processor and can be collected to build the parse tree.

When the parsing starts for a sentence of length $l \leq n+1$, the processors $P1$ to $Pl-1$ are first activated and the processors Pl to Pn are deactivated (i.e. not used for that parsing). It is the task of the **G-CTRL** to activate or deactivate the processors, based on the length l of the sentence. The **G-CTRL** also synchronizes the processors at the end of iteration j (line 7 of the CYK algorithm), before starting iteration $j+1$. This is necessary due to the data dependency among the sets $N_{i,j}$.

The CYK memory stores the sets $N_{i,j}$ and is shared for read and write by all working processors in the system. A token passing priority arbiter handles concurrent accesses to this memory.

The grammar memories store identical copies of the binary representation of the CNF CF grammar. During parsing, the processors intensively access the grammar memories but due to physical constraints (i.e. the number of I/O pins of an FPGA) it is impossible to have a grammar memory for each processor in the system. Instead, processors are grouped in clusters that share the same grammar memory. The number of processors in a cluster may vary from one cluster to the other, and clusters are built in such a way that, in the general case, the number of concurrent accesses is as reduced as possible.

As for the CYK memory, a token passing priority arbiter handles concurrent accesses in every cluster.

3.1 Processor Datapath Structure

The processors have the task of filling-up the entries of the CYK table. More precisely, if l is the sentence length, the task of the processor Pi, $i \leq l-1$, is to fill the CYK table entries on column i during $l-i$ iterations in a bottom-up order. In other words, to compute the sets $N_{i,j}$, based on previously constructed sets (line 10 of the CYK algorithm). The newly computed sets $N_{i,j}$ are stored in the CYK memory for later use. The parsing terminates after exactly $l-1$ iterations, when processor $P1$ finishes to fill-up the column 1.

During iteration j $(2 \leq j \leq l)$, the processors Pi $(1 \leq i \leq l-j+1)$ work in parallel, and the processor Pi is constructing the set $N_{i,j}$. At the end of each iteration the processors wait for a synchronization signal raised by the **G-CTRL** before beginning iteration $j+1$ and the rightmost active processor is deactivated and becomes idle for the rest of the processing time.

The datapath structure of the processor is given in figure 5. The processor is a high frequency pipeline implementation. As depicted in figure 5, we can distinguish 4 main functional units in the processor:

- the CYK memory addressing unit: used by the processor to address the sets $N_{i,j}$ and other data stored in the CYK memory (see section 3.2);
- the update unit: updates the representation of $N_{i,j}$ when necessary (see section 3.3 for details);
- the grammar look-up unit: for computing a new set $N_{i,j}$ a processor requires intensive grammar look-up. This is the task of the **MAG** module. Considering processor Pi during iteration j, the **MAG** module has as input all ordered pairs of

[4] ESL at http://www.embeddedsol.com/tech_info_3.htm

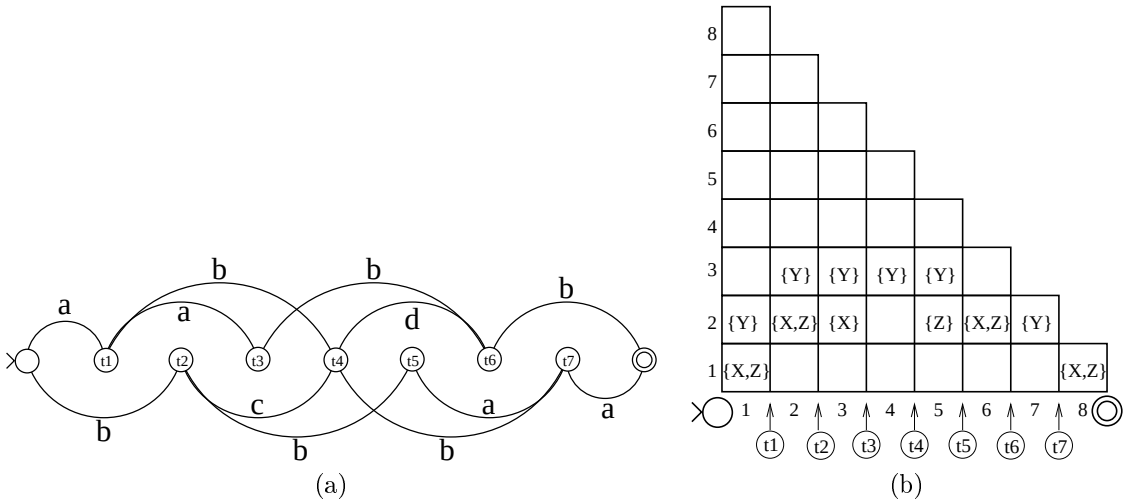

Figure 3: (a) Word-lattice of fig 2 represented as a speech word-lattice (nodes are naturally ordered since they represent different time-instants). (b) The representation of the word-lattice in (a) in the form of a initialized CYK table

non-terminals $(RHS1, RHS2) \in N_{i,k} \times N_{i+k,j-k}$, $1 \leq k \leq j-1$, and gives as output all non-terminals LHS for which there is a rule $LHS \rightarrow RHS1\ RHS2$ in the grammar (see section 3.4 for details);

- the synchronization unit: used by a processor to achieve synchronization with the other processors after each iteration of the CYK algorithm.

3.2 CYK table representation in memory

The purpose of the CYK memory is to store the sets $N_{i,j}$. The data-structure used to represent these sets is critical and also has to correspond to a good compromise between memory size and access-time to the non-terminals in a set. $N_{i,j}$ can be any subset of N, hence $|N_{i,j}|$[5] can be equal to $|N|$ in the worst case. However, to allocate for each set $N_{i,j}$ an amount of memory proportional to $|N|$ would represent an important memory waste, as in practice $|N_{i,j}| \ll |N|$. Therefore, in order to reduce the size of the CYK memory, we impose to $|N_{i,j}|$ an upper limit C. During run-time, if $N_{i,j}$ receives more than C non-terminals, the hardware generates a fault signal and the parsing stops. This is, however, a very unlikely event for a well chosen value of C and we can thus use tables of size proportional to C to store the non-terminals of the sets $N_{i,j}$. Note that, if the number of processors in the system is n, the CYK memory has to store $n(n-1)/2$ such tables.

Given i, j and k, during parsing, a processor has to implement the following three functionalities:

- **F1**: go through all non-terminals of the set $N_{i,k}$ (or $N_{i+k,j-k}$)
- **F2**: is X in $N_{i,j}$?
- **F3**: insert X in $N_{i,j}$

For understanding how these functions are implemented, we give in figure 6 the CYK table organization in memory. We discuss each of these functionalities in turn.

For the implementation of **F1**, the `Phead` pointer is used as a base address that points to the first non-terminal in the non-terminal table. It is stored either in `RHS1base` ($N_{i,k}$) or `RHS2base` ($N_{i+k,j-k}$) registers (see figure 5). A displacement, i.e. index, for addressing any non-terminal stored in the non-terminal table is kept in `RHS1Index`, respectively `RHS2Index`. The addition of the base address with the index address gives the physical address in memory of the non-terminal. For implementing **F1**, the processor needs to know whether the table is empty or not, and, in the later case, to know which is the last non-terminal in the table. This is implemented by means of two special bits attached to each non-terminal in the table. One bit is set when the table is empty, the other when the non-terminal is the last in the table. The current implementation uses 2 bytes for representing

[5] $|N|$ represents the cardinal (the size) of set N.

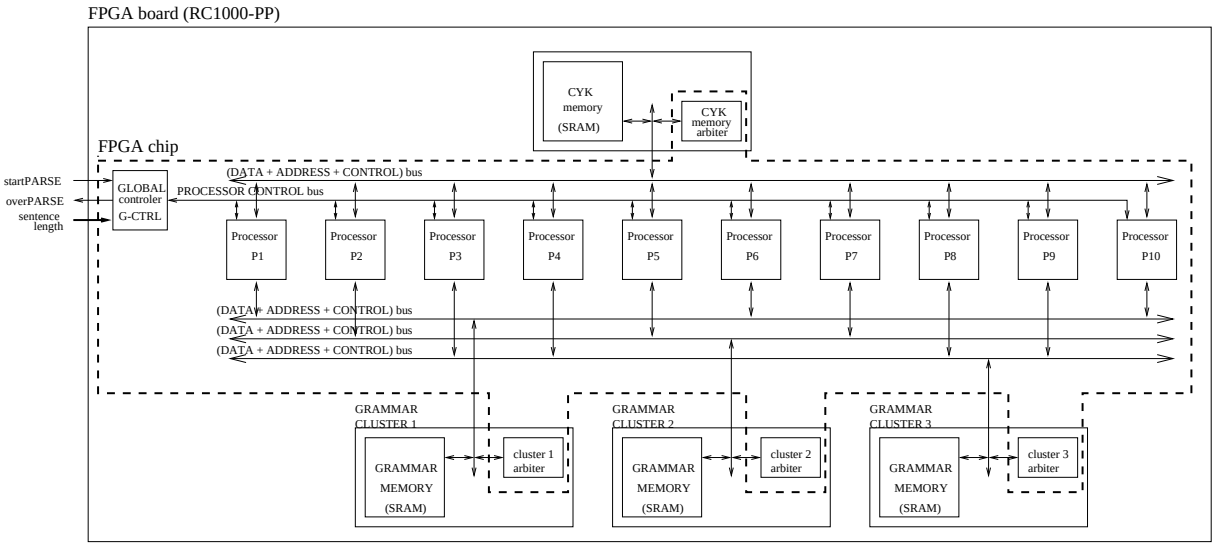

Figure 4: A 10 processor system

a non-terminal, thus, $2C$ bytes are used to represent an entry, i.e. a set, in the CYK table. While 2 bits out of 16 representing the non-terminal are used for special purposes, only 14 are used to code the non-terminal. For this reason only grammars having less than $2^{14} = 16,384$ non-terminals can be considered.

F2 is a highly time consuming function. To speed it up, we use a guard-vector (i.e. bit-vector) table stored in the CYK memory along with the non-terminals table. Every non-terminal in N has an associated bit in the guard-vector which is thus of size $|N|$ bits. If the bit associated with X is set in the guard-vector then X is already in $N_{i,j}$. For the implementation of **F2** the `Pguard` pointer is used as a base address that points to the first bit (in a 4-byte word) in the guard-vector. It is stored in the `Guardbase` register (see figure 5). The displacement for addressing the bit associated to a non-terminal X is given by the binary representation of X.

For the implementation of **F3**, both `Phead` and `Dtail` are used to point the beginning of the non-terminals table, and, respectively, to index the last non-terminal. `Phead` is stored in `IJbase` and `Dtail` in `IJindex` registers. If during the parsing `Dtail` $> C$, a fault signal is raised to signal that the $N_{i,j}$ has too many non-terminals. In a non word-lattice parsing (i.e. sentence parsing) `Dtail` is always 0, and is not used since every set $N_{i,j}$ is empty at the beginning. However, in a word-lattice parsing the sets $N_{i,j}$ are not necessarily empty and the insertion of new non-terminals has to be made at the end of the non-terminals table where `Phead` + `Dtail` points.

The initialization of the CYK table corresponds to: the initialization of the non-terminals table, the associated `Dtail` indexes and the guard-vectors. The `Phead` and `Pguard` pointers are initialized only once and they do not change.

In order to retrieve the pointers `Phead` and `Pguard` and the displacement `Dtail`, the processor builds an address for addressing the indexing table (see figure 6) from i, j and k. The address is constructed in `IJshadow` as $8(32i + j)$ for $N_{i,j}$, in `RHS1shadow` as $8(32i + k)$ for $N_{i,k}$ and in `RHS2shadow` as $8(32(i + k) + j - k)$ for $N_{i+k,j-k}$.

3.3 Update unit

The tasks of the update unit are:

- set the flags (i.e. the 2 special bits) of each non-terminal before writing it in memory in the non-terminal table. This is done in the LHS update module;
- update the guard-vectors. Every time a new non-terminal is inserted in the non-terminal table its corresponding bit in the guard-vector has to be set. This is done in the guard update module.

3.4 Grammar representation in memory

The CYK algorithm uses CF grammars in CNF. Thus, the first pre-processing step is to transform a

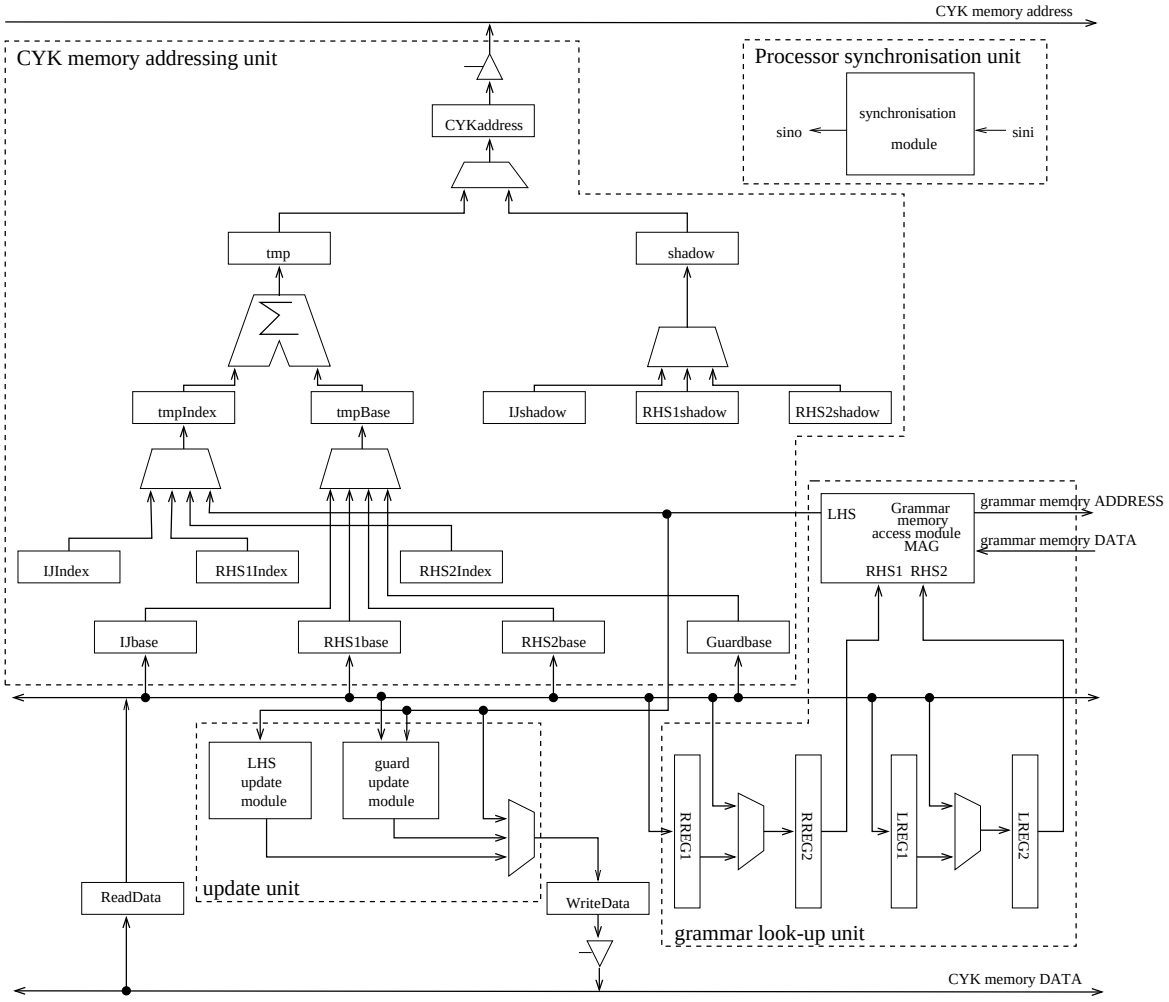

Figure 5: Processor datapath

given general CF grammar in an equivalent CNF CF grammar. This is done off-line only once for each CF grammar used.

The CNF grammar is then represented by a data-structure that has to allow, for any grammar rule right-hand side (RHS) of the form YZ, to retrieve (1) all non-terminals X_i such that there is a rule $X_i \rightarrow YZ$ in the grammar, and (2) a code that uniquely identifies the given RHS. As the data-structure used to represent the grammar is critical for the design, it has to correspond to a good compromise between the memory space taken by and the access time to the stored information.

Concretely, the data-structure representing the grammar is converted in a binary memory image ready to be dumped on the FPGA-board to configure the grammar memories.

As it is shown in figure 7, the data-structure used in our implementation is organized on 3 levels. Level 1 is a table with an entry for each distinct non-terminal present in the grammar. Such an entry contains either a NULL pointer if there is no RHS starting with the corresponding non-terminal Y or a non-NULL pointer pointing to the root of a tree at level 2. Level 2 is a collection of binary sorted trees, each containing all distinct second position non-terminals present in the RHSs that start with a given Y. Finally, in the binary trees, each node contains, in addition to the `ptr_left` and `ptr_right` pointers that link with its left and right sons, the indication of the second position non-terminal Z it corresponds to, and a pointer to a table in level 3 that stores the required information (a RHS code `RHScode` and a list of non-terminals X_1, X_2, ...).

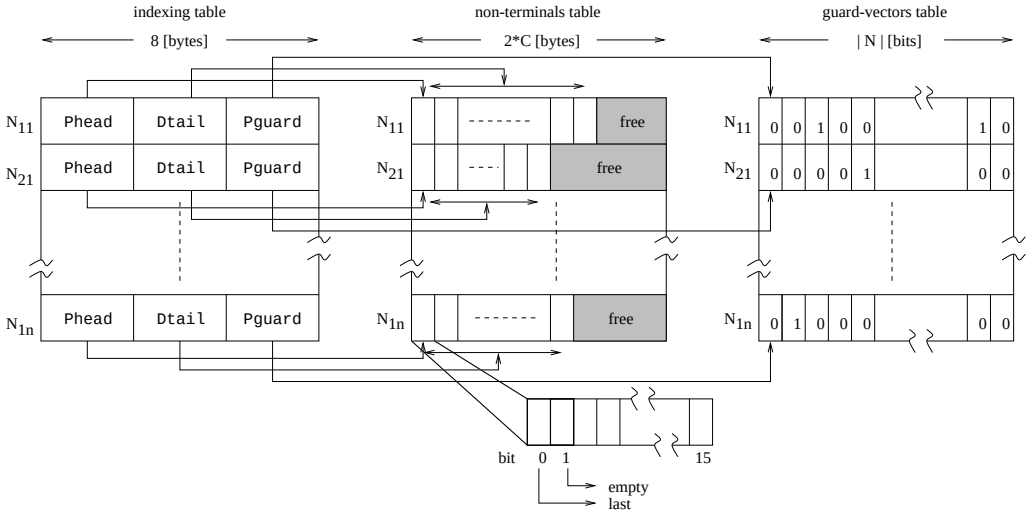

Figure 6: CYK table organization in memory

3.5 Scalability and extensibility properties

Scalability: when the binary memory image for the grammar and CYK memories are created, several parameters characterizing the grammar and CYK datastructures are also computed. These parameters are the number of bits needed to represent a non-terminal, the grammar memory address, the CYK memory address and the RHS code. These parameters are then used to configure the VHDL code. Such a parametric approach allows to scale the design, i.e. to assign to the hardware resources (registers, counters, multiplexers, etc.) bit sizes that match the grammar characteristics. Due to this scalability property, the FPGA resources can be optimally used and, for each grammar, an optimal number of processors can be fit in the FGPA.

Extensibility: in the case where the maximum length of a sentence to be parsed requires a number of processors that does not fit on a single FPGA, the system can be extended by cascading several FPGA circuits. Therefore, the number of processors can be increased as needed and the system can be adapted to parse sentences of any length.

4 Design Performance

All tests and performance measurements presented in this section were made with a grammar extracted from the SUSANNE corpus, referred henceforth as the SUSANNE grammar. In CNF the SUSANNE grammar contains 10,129 non-terminals and 74,350 rules. The grammar memory size required to store the datastructure representing the CNF SUSANNE grammar is of 558,576 bytes. The CYK memory size depends on the number of processors in the system (e.g. 446,496 bytes for a 10 processors system).

In order to determine the real maximal clock frequency at which the system is able to work, the 10 processor system shown in figure 4 was synthesized[6] and placed&routed[7] in a Xilinx FPGA, Virtex V1000bg560-4. The synthesized 10 processor system uses less than 35% of the FPGA resources. The system was then tested, and checked for correctness, on a RC1000-PP board with a clock frequency of 48 MHz.

Due to the fact that the RC1000-PP board can accommodate only 3 grammar clusters, the hardware run-times we present were obtained by simulating[8] the VHDL model of a system with 14 processors and 7 grammar memory clusters.

The software used for comparison is an implementation of an enhanced CYK algorithm developed in our laboratory [2]. The hardware performance (i.e. the run-time hereafter denoted by *hard*) was compared against two software run-times. The former (*soft1*) uses the SUSANNE grammar [12] in CNF, as it is also the case for the hardware. The latter (*soft2*) uses the SUSANNE grammar in its original context-free form. The software was run on a SUN (Ultra-Sparc 60)

[6] With LeonardoSpectrum v1999.1f
[7] With Design Manager (Xilinx Alliance Series 2.1i)
[8] With ModelSim EE/Plus 5.2e

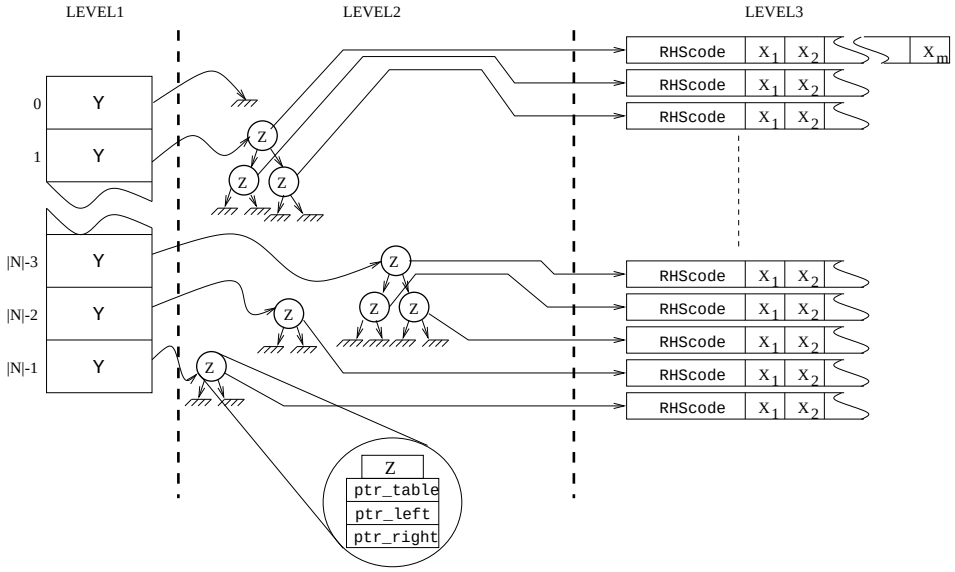

Figure 7: The data-structure used to represent the CNF grammar

with 512 Mbytes memory, 770 Mbytes of swap memory, and 1 processor at a clock frequency of 360 MHz. The initialization of the CYK table was not taken into account for the computation of the run-times. For accuracy, the timing was done with the `time()` C library function and not by profiling the code. For the purpose of the comparison, 2,044 sentences were parsed and validated by comparing the hardware output against the software output for detecting mismatches. The sentences have a length ranging from 3 to 15 and were all taken from the SUSANNE corpus. Figure 8 shows the average run-times *soft1*, *soft2* and *hard* as functions of the sentence length (vertical axe). The average speedup factor $E(S)$ has been computed for both *soft1* and *soft2*. For *soft1*, $E_{soft1}(S) = 69.646$ and for *soft2*, $E_{soft2}(S) = 10.772$. Figure 9 shows the hardware speedup in comparison with *soft1* and *soft2* as a function of the sentence length.

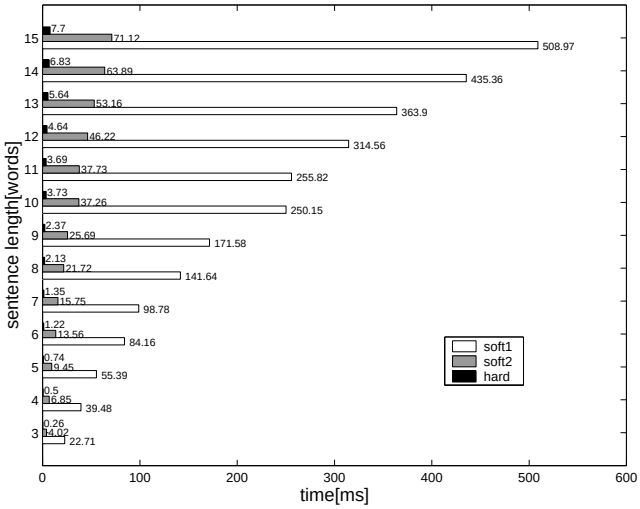

Figure 8: *soft1*, *soft2* and *hard* average run-times

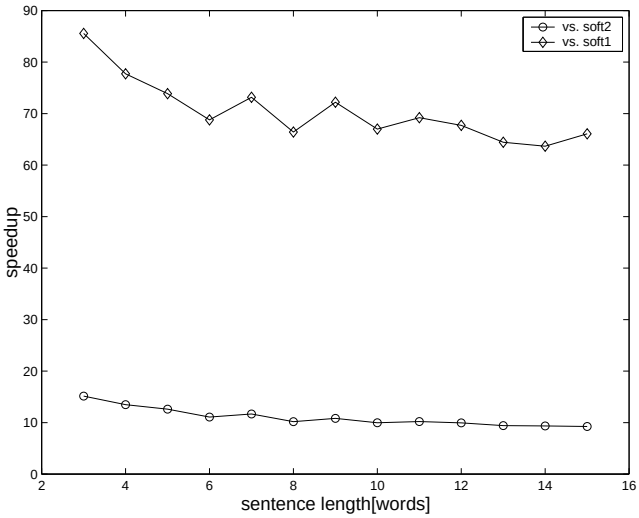

Figure 9: Hardware speedup

5 Conclusions

In this paper we presented an FPGA-based coprocessor implementation of the CYK algorithm adapted for word-lattice parsing and that can accommodate real-life CF grammars. The design interface was designed to allow an easy integration of the parser within a larger system (e.g. a speech-recognition application on a desktop computer) in which the parsing hardware would work as a co-processing unit.

The performance measurements show an average speedup of 10.7 for the hardware when compared with our best software implementation of the CYK algorithm using a general CF grammar and a speedup of 69.9 when compared with the software using a CNF version of the same grammar.

These preliminary experiments represent encouraging results for the application of the reconfigurable computing paradigm for the implementation of NLP algorithms and other applications requiring efficient parsing with CF grammars.

In addition, the experience acquired during the implementation of the system suggests the following possible extensions for our research work:

- further improvements of the design such as a better processor control corresponding to a higher exploitation of the parallelism available in the CYK algorithm and therefore to a more efficient processor utilization. In particular, the average number of processors idle during parsing should be substantially reduced;
- further functional extensions of the design such as stochastic, i.e. probabilistic, parsing and the ability for the design to cope with general CF grammar not requiring a preliminary transformation in CNF.

References

[1] A. V. Aho and J. D. Ullman. *"The Theory of Parsing, Translation and Compiling"*, volume 1. Prentice-Hall, 1972.

[2] J.-C. Chappelier and M. Rajman. A generalized CYK algorithm for parsing stochastic CFG. In *TAPD'98 Workshop*, pages 133–137, Paris (France), 1998.

[3] Y. T. Chiang and K.S. Fu. "Parallel Parsing Algorithms and VLSI Implementations for Syntactic Pattern Recognition". In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, May 1984.

[4] K.H. Chu and K.S. Fu. "VLSI architectures for high speed recognition of context-free languages and finite-state languages.". In *Proc. 9th Annu. Int. Symp. Comput. Arch.*, pages 43–49, April 1982.

[5] J.-C. Chappelier et al. Lattice parsing for speech recognition. In *Proc. of 6ème conférence sur le Traitement Automatique du Langage Naturel (TALN'99)*, pages 95–104, Cargèse (France), July 1999.

[6] R. Feldman et al. Text mining at the term level. In *Proc. of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD'98)*, Nantes, France, Sept 1998.

[7] R. Feldman, I. Dagan, and W. Kloegsen. Efficient algorithm for mining and manipulating associations in texts. In 13^{th} *European Meeting on Cybernetics and Research*, 1996.

[8] D. et al. Hull. Xerox trec-5 sire report: Routing, filtering, nlp and spanish tracks. In *NIST Special Publication 500-238: The Fifth Text REtrieval Conference (TREC-5)*, Gaithersburg, Maryland, November 1996.

[9] S. R. Kosaraju. "Speed of recognition of context-free languages by array automata". *SIAM J. Comput.*, 4:331–340, 1975.

[10] Fu K.S. *Syntactic methods in pattern recognition.* Academic Press, 1974.

[11] Linz P. *An Introduction to Formal Languages and Automata.* Jones and Bartlett Publishers, 1997.

[12] G. Sampson. "The Susanne Corpus Release 3". School of Cognitive & Computing Sciences, University of Sussex, Falmer, Brighton, England, 1994.

[13] P. Schäuble. *Mutlimedia Information Retrieval - Content-Based Information Retrieval from Large Text and Audio Databases.* Kluwer Academic Publishers, 1997.