Instance-Specific Solutions to Accelerate the CKY Parsing *

Jacir L. Bordim
Adaptive Communications Research Laboratories
ATR
2-2-2 Hikaridai, Kyoto 619-0288 Japan

Yasuaki Ito
School of Information Science
JAIST
Tatsunokuchi, Ishikawa 923-1292, Japan

Koji Nakano
Department of Artificial Complex Systems Engineering
Hiroshima University
Kagamiyama, Hirhoshima 739-8527, Japan

Abstract

The main contribution of this paper is to present an FPGA-based implementation of an instance-specific hardware which accelerates the CKY (Cocke-Kasami-Younger) parsing for context-free grammars. Given a context-free grammar G and a string x, the CKY parsing determines whether G derives x. We developed a hardware generator that creates a Verilog HDL source to perform the CKY parsing for any given context-free grammar G. The generated source is embedded in an FPGA using the design software provided by the FPGA vendor. The results show that our instance-specific hardware solution attains an astonishing speed-up factor of up to 3,700 over traditional software solutions.

keywords: CKY Parsing, FPGAs, Reconfigurable architectures, Reconfigurable computing.

1 Introduction

An FPGA (Field Programmable Gate Array) is a programmable VLSI in which a hardware design can be embedded instantly. Typical FPGAs consist of an array of programmable logic elements, distributed memory blocks, and programmable interconnections between them. The logic block usually contains either a two-input logic function or a 4-to-1 multiplexer and several flip-flops. The distributed memory block is usually a dual-port RAM on which a word of data for possibly distinct addresses can be read/written at the same time. Using the design tools supplied by the FPGA vendors, the user hardware logic design can be embedded into the FPGAs. Our goal is to use FPGAs to accelerate useful computations. In particular, the challenge is to develop FPGA-based solutions that are faster and more efficient than traditional software solutions.

The approach of accelerating computations using FPGAs is inspired by the notion of *partial computation* [10]. Let $f(x, y)$ be a function to be evaluated in order to solve a given problem. Note that such a function might be repeatedly evaluated only for a fixed x. When this is the case, the computation of $f(x, y)$ can be simplified by evaluating an instance-specific function f_x such that $f_x(y) = f(x, y)$. For instance, imagine a problem such that an algorithm to solve it evaluates $f(x, y) = x^3 + x^2y + y$ repeatedly. If $f(x, y)$ is evaluated only for $x = 2$, then the formula can be simplified such that $f_2(y) = 8 + 5y$. The optimization of function f_x for a particular x is called a *partial computation*.

Our novel idea is to build hardware solutions that are optimized to compute $f_x(y)$ for a fixed x and various y. More specifically, the goal is to present instance-specific solutions for problems that involves a function evaluation for $f(x, y)$ satisfying the following properties:

1. The value of a fixed instance x depends on the instance of the problem, and
2. The value of $f(x, y)$ is repeatedly evaluated for various y to solve the problem.

The main contribution of this paper is to present instance-specific solutions to accelerate the parsing for context-free grammars [13]. Let $f(G, x)$ be a function such

*Work supported in part by the Ministry of Education, Science, Sports, and Culture, Government of Japan, Grant-in-Aid for Exploratory Research (14658090)

that G is a context-free grammar, x is a string, and $f(G, x)$ returns a Boolean value such that $f(G, x)$ returns TRUE iff G derives x. It is well-known that the *CKY (Cocke-Kasami-Younger) parsing* [1] computes $f(G, x)$ in $O(n^3)$ time, where n is the length of x [1]. The parsing of context-free languages has many application in various areas including natural language processing [5, 15], compiler construction [1], informatics [14], among others.

Several studies have been devoted to accelerate the parsing of context-free languages [3, 4, 9, 12, 15]. It has been shown that the parsing for a string of length n can be done in $O((\log n)^2)$ time using n^6 processors on the PRAM [9]. Also, using the mesh-connected processor arrays, the parsing can be done in $O(n^2)$ time using n processors as well as in $O(n)$ time using n^2 processors [12]. Since these parallel algorithms need at least n processors, they are unrealistic for large n. Ciressan *et al.* [6, 7] have presented a hardware for the CKY parsing for a restricted class of context-free grammar and have tested it using FPGA. However, the hardware design and the control algorithm are essentially the same as those on the mesh-connected processors [12], and they are not instance-specific.

As pointed out by Ninomiya *et. al.* [11], recent Natural Language Processing has received a great deal of attention from the research community, however a limited number of works are beyond theoretical speculation or experiments with small grammars. Also in [11], the authors proposed a parallel implementation to parse grammars with a significant number of rules $(18, 891)$ and non-terminal symbols (206). To show the applicability of their results, the proposed implementation has been used to parse a report from the Ministry of Internal Trade and Industry (report written in Japanese). Quite recently, we presented an FPGA-based implementation to accelerate the CKY parsing and tested it on the Altera APEX20K series FPGA [3]. This implementation achieved an speed-up factor of nearly 750 over traditional software approaches with rules ranging from 32 to 8,192, input string length of 32, and with both 32 and 64 non-terminal symbols. Although our results provided a significant improvement over traditional approaches, we have been able to verify them only for small input sequences and a constrained number of production rules. The main limitation of our earlier approach was the need to store computed values in the FPGA memory cells, which is impractical for large input sequences.

For the purpose of instance-specific solution for parsing context-free languages, we present a hardware generator that produces a Verilog HDL source that performs the CKY parsing for any given context-free grammar G. The key ingredient of the produced design is a hardware component to compute a binary operator $\otimes_G$ such that $2^N \times 2^N \rightarrow 2^N$, where N is the set of non-terminal symbols in G. More specifically, let U and V be a set of non-terminals in G that derive strings α and β, respectively. The operator $U \otimes_G V$ returns the set of non-terminals that derive $\alpha\beta$ (i.e. the concatenation of α and β). The CKY parsing algorithm repeats the evaluation of $\otimes_G$ for $O(n^3)$ times. The details of $\otimes_G$ will be explained in Section 2.

In order to compute all possible derivations of the input string x, the CKY algorithm uses a two-dimensional array, called the CKY Table. In this work we consider two different hardware approaches to speed-up the computation of the CKY algorithm. In the first approach, both the component for computing $\otimes_G$ and the CKY table are implemented in FPGAs. In the second approach, the component for computing $\otimes_G$ is implemented in FPGA while the CKY table is stored in the Host-PC's main memory. When evaluating $\otimes_G$, the necessary information is transferred from the Host-PC to the FPGA via the PCI-bus. Note that today's PCs can be embedded with Gbyte-memory size while even the most advanced FPGAs are embedded with only Mbyte-memory size. Thus, the second approach allows us to handle larger input strings. We have verified our implementations for rules ranging from 5,000 to 25,000 using either 256 or 512 non-terminal symbols with the length of the input string ranging from 90 to 512. Clearly, our results are good enough to parse a corpus like the one presented in [11].

Figure 1 illustrates our CKY Hardware Parsing System. The CKY Hardware Parsing System takes the context-free grammar G as input and generates a Verilog HDL source file of the CKY parser. The Verilog HDL source is compiled using the ISE Logic Design Tool [17] and the object file obtained is downloaded into the Xilinx Virtex-II series FPGAs [16]. The programmed FPGA compute $f_G(x)$, i.e. determines if G derives x for a given string x. Given strings $x_1, x_2, x_3, \ldots$ by the host PC, the FPGA computes and returns $f_G(x_1), f_G(x_2), f_G(x_3), \ldots$ to the host.

Traditional sequential software approaches compute $\otimes_G$ by checking all p production rules in $O(p)$ time. Hence, a sequential software implementation of the CKY parsing algorithm runs in $O(n^3 p)$ time. The proposed instance-specific hardware solution evaluates $\otimes_G$ in $O(\log b)$ time. Thus, the CKY parsing can be computed in $O(n^3 \log b)$ time. Since $b \leq p$ always hold, our hardware solution is much faster, from the theoretical point of view, than the traditional software approaches.

We have implemented and evaluated the performance of our instance-specific hardware solution on the Xilinx Virtex-II Series FPGA. To evaluate the performance of our hardware solution we provided a traditional software implementation as counterpart. The performance evaluation has been carried out on an IBM PC-compatible (Xeon 2.8GHz processor). The results show that our instance-specific hardware solution attains an astonishing speed-up factor of nearly 3,700 over traditional software solutions.

This paper is organized as follows: Section 2 briefly de-

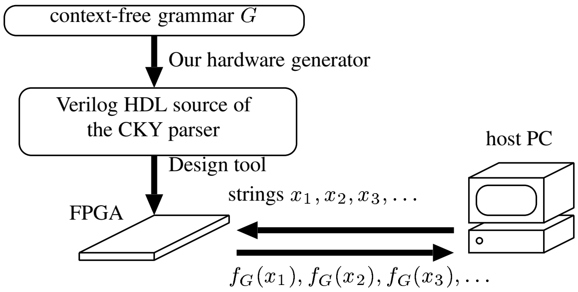

Figure 1. The CKY Hardware Parsing System.

scribes the CKY parsing scheme and a traditional software implementation. In Section 3, we present the details of our instance-specific hardware solutions for the CKY parsing. Section 4 evaluates the performance of our instance-specific hardware solutions and compare the obtained results to the software solutions. Finally, Section 5 concludes this work.

2 The CKY parsing and software solutions

This section briefly describe the CKY parsing and a traditional software solution. Let $G = (N, \Sigma, P, S)$ denote a *context-free grammar* such that N is a set of non-terminal symbols, Σ is a set of terminal symbols, P is a set of production rules, and S $(\in N)$ is the start symbol. A context-free grammar is said to be in *Chomsky Normal Form* (CNF), if every production rule in P is in either form $A \rightarrow BC$ or $A \rightarrow a$, where A, B, and C are non-terminal symbols and a is a terminal symbol.

We are interested in the parsing problem for a context-free grammar in CNF. More specifically, for a given CNF context-free grammar G and a string x over Σ, the parsing problem asks to determine if the start symbol S derives x. For example, let $G_{\text{example}} = (N, \Sigma, P, S)$ be a grammar such that $N = \{S, A, B\}$, $\Sigma = \{a, b\}$, and $P = \{S \rightarrow AB, S \rightarrow BA, S \rightarrow SS, A \rightarrow AB, B \rightarrow BA, A \rightarrow a, B \rightarrow b\}$. The context-free grammar G derives $abaab$, because S derives it as follows:

$$S \Rightarrow AB \Rightarrow ABA \Rightarrow ABAA \Rightarrow ABAAB \Rightarrow \cdots \Rightarrow abaab.$$

We are going to explain the *CKY parsing scheme* that determines whether G derives x for a CNF context-free grammar G and a string x. Let $x = x_1 x_2 \cdots x_n$ be a string of length n, where each x_i $(1 \le i \le n)$ is in Σ. Let $T[i, j]$ $(1 \le i \le j \le n)$ denote a subset of N such that every A in $T[i, j]$ derives a substring $x_i x_{i+1} \cdots x_j$. The idea of the CKY parsing is to compute every $T[i, j]$ using the following relations:

$$T[i,i] = \{A \mid (A \rightarrow x_i) \in P\}$$

$$T[i,j] = \bigcup_{k=i}^{j-1} \{A \mid (A \rightarrow BC) \in P, B \in T[i,k], \text{ and } C \in T[k+1,j]\}$$

A two-dimensional array T is called the *CKY table*. A grammar G generates a string x iff S is in $T[1, n]$. Let $\otimes_G$ denote a binary operator $2^N \times 2^N \rightarrow 2^N$ such that $U \otimes_G V = \{A \mid (A \rightarrow BC) \in P, B \in U, \text{ and } C \in V\}$. The details of the CKY parsing are spelled out as follows:

CKY parsing
1. $T[i, i] \leftarrow \{A \mid (A \rightarrow x_i) \in P\}$ for every i $(1 \le i \le n)$
2. $T[i, j] \leftarrow \emptyset$ for every i and j $(1 \le i < j \le n)$
3. for $j \leftarrow 2$ to n do
4. for $i \leftarrow j - 1$ downto 1 do
5. for $k \leftarrow i$ to $j - 1$ do
6. $T[i, j] \leftarrow T[i, j] \bigcup (T[i, k] \otimes_G T[k + 1, j])$

The first two lines initialize the CKY table, and the next four lines compute the CKY table. Figure 2 illustrates the CKY table for G_{example} and the string $abaab$. Since $S \in T[1, 5]$, one can see that G_{example} derives $abaab$.

Clearly, the last four lines are dominant in the CKY parsing. Let t be the computing time necessary to perform an iteration of the line 6. Then, line 6 is executed for

$$\sum_{j=2}^{n-1}\sum_{i=1}^{j-1}\sum_{k=i}^{j-1} t = t\sum_{j=2}^{n-1}\sum_{i=1}^{j-1}(j-i) = \frac{1}{6}t(n^3 - 3n^2 + 2n)$$

times.

Let us evaluate the computing time t necessary to perform line 6, i.e., necessary to evaluate the binary operator $\otimes_G$. A traditional software approach (i.e, sequential algorithm), checks whether $B \in U$ and $C \in V$ for every production rule $A \rightarrow BC$ in P. Clearly, using a reasonable data structure, this can be done in $O(1)$ time. Hence, $U \otimes_G V$ can be evaluated in $O(p)$ time, where p is the number of production rules in P that has the form $A \rightarrow BC$. Thus, using the above approach, the CKY parsing can be computed in $O(n^3 p)$ time. The performance evaluation of our hardware implementation will use the aforementioned software solution as counterpart in the performance evaluation (Section 4).

		i				
		1	2	3	4	5
j	5	S, A	S, B	$\emptyset$	S, A	B
	4	S, A	S, B	$\emptyset$	A	b
	3	S, A	S, B	A	a	
	2	S, A	B	a		
	1	A	b			
		a				

Figure 2. The CKY table for G_{example} and $abaab$.

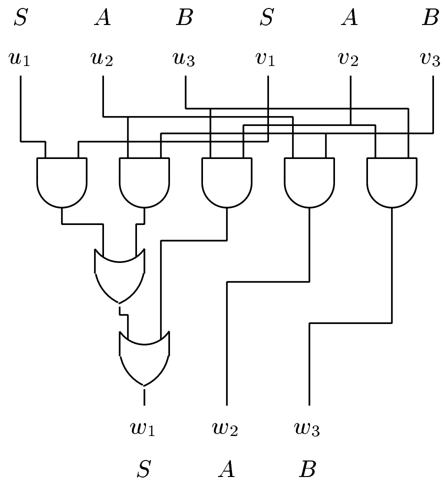

Figure 3. The circuit for computing $\otimes_{G_{\text{example}}}$.

3 Our instance-specific hardware for the CKY parsing

This section is devoted to show our instance-specific hardware for the CKY parsing. We first accelerate the evaluation of $\otimes_G$ by building a circuit for computing $\otimes_G$ in an FPGA. We then go on to show the hardware details to build this circuit. Next, we present the details of a second hardware implementation in which the CKY table is stored in the Host-PC.

Let U and V ($\in 2^N$) be represented by b-bit binary vectors $u_1u_2\cdots u_b$ and $v_1v_2\cdots v_b$, respectively, where b represents the number of non-terminal symbols. Our goal is to compute the vector $w_1w_2\cdots w_b$, which represents $W = U \otimes_G V$. For a particular w_k, we are going to show how w_k is computed. Let $N_k \rightarrow N_{i_1}N_{j_1}$, $N_k \rightarrow N_{i_2}N_{j_2}$, …, and, $N_k \rightarrow N_{i_s}N_{j_s}$ be the production rules in P whose non-terminal in the left-hand side is N_k. Then, w_k is computed by the following formula:

$$w_k \quad = \quad (u_{i_1} \wedge v_{j_1}) \vee (u_{i_2} \wedge v_{j_2}) \vee \cdots \vee (u_{i_s} \wedge v_{j_s}).$$

The task of our hardware generator is to read the production rules in P, which are stored in a text file, and generate a *module* to compute the vector $w_1w_2\cdots w_b$. Based on the production rules, our hardware generator creates the necessary code to compute each entry w_k. A module in Verilog-HDL is analogue to a procedure in a high-level language, such as C/Pascal, and can be "called" from the main module. The main module comprehend a number of functions, whose tasks are, among others, to control memory access and the FPGA-PC interface. An example of the code created by our hardware generator is shown below.

```
1 module comp(u,v,w);
2   input [3:1] u,v;
3   output [3:1] w;
4
5   assign w[1] = (u[2] & v[3])
6               | (u[3] & v[2])
7               | (u[1] & v[1]);
8   assign w[2] = (u[2] & v[3]);
9   assign w[3] = (u[3] & v[2]);
10 endmodule
```

The first line defines the module name and the parameters received and returned by the module. The parameters are explicit defined as shown in lines 2 and 3. Each entry of the output vector is computed in lines 5 trough 9, which are computed according to the production rules in P. The circuit of the above module is shown in Figure 3. As shown above, w_k can be computed by a combinatorial circuit using s AND-gates and $s-1$ OR-gates with fan-in 2. Furthermore, the depth of the circuit (or the maximum number of gates over all paths in the circuit) is $\lceil \log(s-1) \rceil + 1$. Since we have p production rules of the type $A \rightarrow BC$ in P, then $w_1w_2\cdots w_b$ can be computed by a circuit with p AND-gates and $p-b$ OR-gates. Because $s \le b^2$ always hold, the depth of the circuit is no more than $\lceil \log(b^2-1) \rceil + 1 \le 2\log b + 1$. Thus, the CKY parsing can be done in $O(n^3 \log b)$ time using this circuit. Figure 3 illustrates a circuit for $\otimes_{G_{\text{example}}}$. Since G_{example} has 5 production rules and 3 non-terminal symbols, the circuit has 5 AND gates and $5-3=2$ OR gates.

The sequential algorithm we have discussed in Section 2 takes $O(p)$ time to evaluate $\otimes_G$. On the other hand, our circuit for $\otimes_G$ has a delay time proportional to $O(\log b)$.

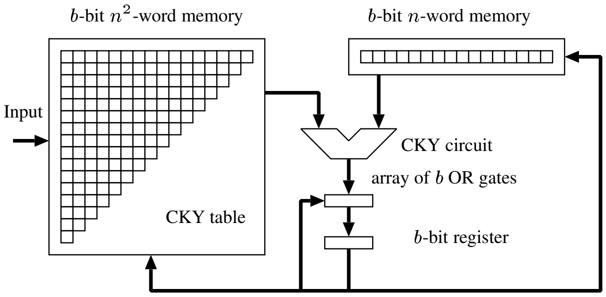

Figure 4. A hardware implementation for the CKY parsing.

Since $b \leq p$ always holds, the circuit for $\otimes_G$ is faster than the sequential algorithm from the theoretical point of view. In what follows, we are going to show the implementation details of our instance-specific hardware. Our first hardware implementation of the CKY parsing uses the following basic components:

– a b-bit n^2-word (dual-port) memory;

– a b-bit n-word (dual-port) memory;

– a CKY circuit for $\otimes_G$;

– an array of b OR gates; and

– a b-bit register.

Figure 4 illustrates our first implementation for the CKY parsing. The b-bit n^2-word memory stores the CKY table. The input, $T[1,1]$, $T[2,2]$, $\ldots, T[n,n]$ is supplied to the b-bit n^2-word memory. The b-bit n-word memory stores a row of the CKY table that is being processed. In other words, it stores the j-th row $T[1,j], T[2,j], \ldots$ of the CKY table, where j is the variable appearing in line 3 of the CKY parsing. The b-bit register stores the current value of $T[i,j]$, which is computed in line 6 of the CKY parsing. The array of b OR gates is used to compute "$\bigcup$" in line 6. The b-bit n^2-word memory supplies the b-bit vector representing $T[i,k]$ to the CKY circuit. Similarly, the b-bit n-word memory outputs the b-bit vector for $T[k+1,j]$. The CKY circuit receives them and computes the b-bit vector for $T[i,k] \otimes_G T[k+1,j]$. Using this hardware implementation, line 6 of the CKY parsing is computed in a clock cycle. Thus, the CKY parsing can be done in n^3 clock cycles. Furthermore, in a real implementation, a clock cycle is proportional to $O(\log b)$. Thus, the computing time is $O(n^3 \log b)$.

In the above approach, a large portion of the FPGA memory blocks is used to store the CKY table. Hence, for large input strings, it might not be feasible to store the CKY table in the FPGA due to limited amount of hardware. A way to overcome this limitation is to store the CKY table in the Host-PC. Our second implementation of the CKY parsing explores this alternative. This implementation uses the same basic components of the first implementation, except for the b-bit n^2-word memory which is now stored in the Host-PC. The details of the second hardware implementation is shown in Figure 5.

When computing the j-th row of the CKY table, all the necessary information is transferred from the Host-PC to the FPGA. The CKY circuit retrieves the information needed to compute each entry $T[i,j]$ in line 6. To be more precise, let us consider the computation of a particular row, say the m-th row. The first element (i.e., the first entry) of the m-th row is provided by the CKY table, since it is a terminal symbol. Such symbol is then stored in the b-bit n-word memory. To compute the next entry $T[i,m]$, the information regarding the already computed entries of the i-th column and m-row are necessary. The elements of the i-th column are provided by the Host-PC via the PCI-bus, while the previously computed elements of the m-th row are provided by the b-bit n-word memory (which is stored in the FPGA). Once the entry $T[i,m]$ is computed, its result is stored in the b-bit n-word memory for later use. Upon completing the computation of the m-row, all the computed elements are transferred to the Host-PC where they are stored. Although the computing time for our second implementation is also $O(n^3 \log b)$ some overhead is introduced due to the use of the PCI-bus.

The subsequent section shown the performance evaluation of our hardware implementations and compare them with traditional software solutions.

4 Performance evaluation

We have implemented and evaluated the performance of our instance-specific solution on a Xilinx Virtex-II series FPGA (XC2V3000, speed grade 4, typical 3 Million gates with 1.7 Mbits embedded memory and 32K logic cells). In order to compare the performance of our instance-specific solution, we have implemented two software solutions and measured the performance on an IBM PC-compatible (Xeon 2.8GHz processor with 2GB Memory) using Linux OS (Kernel 2.2.18-14smp). The software solution has been implemented in C++ and compiled with the Intel C++ compiler for Linux 7.0.

Table 1 shows the performance of our first hardware approach and its corresponding software solution for the CKY algorithm for $b = 32$ and $n = 90$ (recall that n represents the length of the input string and b represents the number of non-terminal symbols). As shown in the table, a speed-up factor of more than 900 is experienced for p = 5,000.

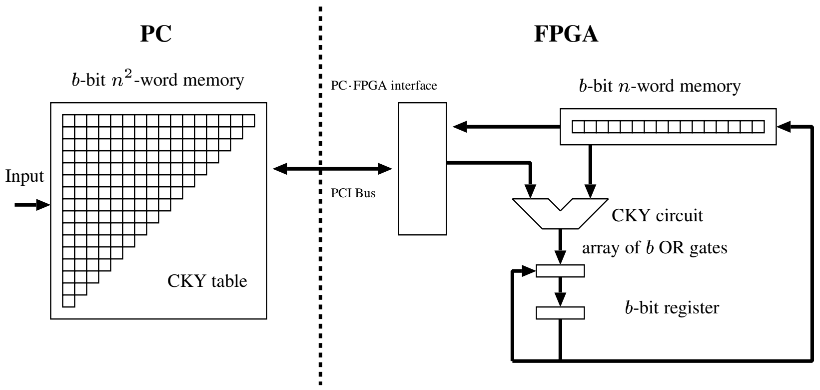

Figure 5. Hardware implementation of the CKY parsing with the CKY table stored in the Host-PC.

n	b	p	FPGA [*ms*]	Software [*ms*]	Speed-up
90	256	5,000	3.109	2,850	917
		10,000		6,050	1,946
		15,000		9,100	2,927
	206	19,000		11,490	3,696

Table 1. Performance of the CKY algorithm with the CKY Table implemented in FPGA.

Recall that the sequential algorithm checks whether or not $B \in U$ and $C \in V$ for every production rule $A \to BC$ in P. Hence, the computing time of the sequential algorithm is proportional to the number of production rules. That is, the computing time of the software solution increases along with the number of production rules. On the other hand, the computing time of the hardware implementation is independent of the number of production rules. For $p = 10{,}000$, the FPGA solution attains an speed-up factor of more than 1,900, and for $p = 15{,}000$ we observed an speed-up factor of nearly 3,000. For $p = 19{,}000$ and $b = 206$ we observed an speed up factor of nearly 3,700. The above instance-specific hardware solution, with the CKY table implemented in the FPGA, utilizes up to 85% of the available memory blocks. This fact prevented us for implementing hardware solutions for larger values of n. To overcome this limitation, we proposed a hardware solution where the CKY table is stored in the Host-PC.

In what follows, we present the results of our second hardware implementation. Table 2 shows the computing time of our second hardware approach along with the computing time of the software solution. The performance evaluation has been carried out for different values of b and p with the input string fixed to 512 (i.e., $n = 512$). The computing time for the CKY implementation in software follows the same pattern observed in the previous table. As expected, the implementation of the CKY table in the Host-PC adds a considerable overhead with has profound impact in the performance of the hardware solution. We have been able to implement our second approach using up to 25,000 rules. We also attempted using 30,000 rules which surpassed the amount of available resources of our FPGA, thus resulting in "routing error". Although the FPGA cycle speed is roughly the same of the PCI-bus (33MHz), we speculate that the time to access the PCI to send/receive data is responsible for a large part of the introduced overhead. Notwithstanding, the hardware implementation with the CKY table stored in the Host-PC attains an speed-up factor of up to 93 over the software approach.

5 Concluding remarks

The main contribution of this work was to present an FPGA-based implementation of an instance-specific hardware that accelerates the CKY parsing for context-free grammars.

We have implemented our instance-specific hardware solution on the Xilinx Virtex-II Series FPGA. To evaluate the performance of our hardware solution we provided a traditional software implementation as counterpart. The performance evaluation has been carried out on an IBM PC-compatible (Xeon 2.8GHz processor). The results have shown that our instance-specific hardware solution attain an astonishing speed-up factor of nearly 3,700 over traditional software solutions.

b	n	p	FPGA [s]	Software [s]	Speed-up
512	128	15,000	0.476	23.54	49
		20,000		33.49	70
		25,000		42.84	90
	256	15,000	3.819	119.05	31
		20,000		276.47	72
		25,000		356.23	93
	512	15,000	30.456	1,653.95	54
		20,000		2,246.00	74
		25,000		2,823.00	93

Table 2. Performance of the CKY algorithm with the CKY Table implemented in the Host-PC

References

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

[2] Altera Corporation, APEX 20K Devices: System-on-a-Programmable-Chip Solutions, http://www.altera.com/products/devices/apex/apx-index.html.

[3] J. L. Bordim, Y. Ito, and K. Nakano, *Accelerating the CKY Parsing Using FPGAs*, Proc. of the 9th International Conference on High Performance Computing (HiPC-2002), pp.41-51, Dec. 18-21, Bangalore, India.

[4] J. Chang, O. Ibarra, and M. Palis, *Parallel parsing on a one-way array of finite-state machines*, IEEE Transactions on Computers, C-36(1):64–75, 1987.

[5] E. Charniak, *Statistical Language Learning*, MIT Press, Cambridge, Massachusetts, 1993.

[6] C. Ciressan, E. Sanchez, M. Rajman, and J.-C. Chappelier, *An FPGA-based coprocessor for the parsing of context-free grammars*, In Proc. of IEEE Symposium on Field-Programmable Custom Computing Machines, 2000.

[7] C. Ciressan, E. Sanchez, M. Rajman, and J.-C. Chappelier, *An FPGA-based syntactic parser for real-life almost unrestricted context-free grammars*, In Proc. of International Conference on Field Programmable Logic and Applications (FPL), pages 590–594, 2001.

[8] Y. Futamura, K. Nogi, and A. Takano, *Essence of generalized partial computation*, Theoretical Computer Science, 90:61–79, 1991.

[9] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms*, Cambridge University Press, 1988.

[10] N. D. Jones, C.K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice Hall, 1993.

[11] T. Ninomiya, K. Toriswa, K. Taura, and Jun-ichi Tsujii, *A Parallel CKY Parsing Algorithm on Large-Scale Distributed-Memory Parallel Machines*, in Proceedings of the Pacific Association for Computational Linguistics '97, pp. 223–231, Tokyo, Japan, September, 1997.

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

[13] J. C. Martin, *Introduction to languages and the theory of computation (2nd Edition)*, Mac-Graw Hill, 1996.

[14] Y. Sakakibara, M. Brown, R. Hughey, I. S. Mian, K. Sjölander, R. C. Underwood, and D. Haussler, *Stochastic context-free grammars for tRNA modeling*, Nucleic Acids Research, 22:5112–5120, 1994.

[15] M. P. van Lohuizen, *Survey on parallel context-free parsing techniques* Technical Report IMPACT-NLI-1997-1, Delft University of Technology, 1997.

[16] Xilinx Inc., Virtex II FPGAs http://www.xilinx.com

[17] Xilinx Inc., ISE Logic Design Tools, http://www.xilinx.com/ise/design_tools/index.html