XML Accelerator Engine

Jan van Lunteren, Ton Engbersen,
IBM Research, Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland

Joe Bostian, Bill Carey, Chris Larsson
IBM
Poughkeepsie, NY 12601, USA

***Abstract*— Emerging trends in information technology create the need for very high XML processing performance, which might only be achievable by applying a non-traditional processor architecture. This paper presents a novel programmable state machine technology, called B-FSM, which can become the key enabler for such an architecture. This technology is fully programmable and provides high performance in combination with low storage requirements and fast incremental updates. Based on the B-FSM technology, the high-level concept of an XML acceleration engine called ZUXA is introduced, which provides a processing model optimized for conditional execution in combination with dedicated instructions for character and string-processing functions.**

I. INTRODUCTION

The Extensible Markup Language (XML) [1] has become a fundamental technology to facilitate the description of information and its interchange within systems and between nodes in a heterogenous computing environment. It plays an ever increasing role in user interfaces such as browser- and voice-based, local program data representations such as for configuration files, various standards definitions such as in the security arena, and program to program communications [2]. Most, if not all, recent standards describing communications between services in a distributed environment are described as XML-based grammars. As a result, any server that produces information in a web services context must understand and generate XML documents, even if these documents only serve as containers for other non-XML data.

As web services continue to evolve and mature as resources that are to be delivered on demand, the XML parsing workload that computing nodes must support is expected to grow at an increasing rate. Database servers, which historically have been the repository for large volumes of information, must process XML data as efficiently as possible or face the threat of being overwhelmed by the XML parsing overhead [3].

There seems to be a fundamental problem with XML processing in software that will prevent the processing rate to be improved beyond a best processing rate of tens of clock cycles per character, and that for many XML applications can result in processing rates on the order of hundreds of clock cycles per character. This performance bottleneck may only be overcome by applying a non-traditional processor architecture. This paper presents a novel programmable state machine technology that can become a key enabler for building such a novel approach. The paper is organized as follows: Section II describes the programmable state machine technology, which is the main topic of the paper; Section III outlines several performance bottlenecks of XML processing in software; Section IV describes how the programmable state machine technology can be used to build a high-performance XML accelerator engine, and Section V concludes the paper.

II. PROGRAMMABLE STATE MACHINE

A. Design Objectives

The novel state machine technology has been designed to meet the following requirements:

1. *High performance*: The objective is to achieve a performance of one state transition per clock cycle for clock frequencies in the 0.5-2 GHz range.
2. *Storage efficiency*: The memory requirements should be small to allow the cost-efficient use of fast on-chip memory technologies.
3. *Programmable*: The state machine has to be fully programmable. It has to support fast incremental updates of the data structure, allowing new states and transitions to be added dynamically without the need for updating the entire data structure.
4. *Wide input and output vectors*: The state machine has to be able to make transitions based on input symbols with a width between 16 and 32 bits, while it has to be able to generate output vectors of at least 64 bits for each transition (Mealy machine).
5. *Scalability*: The state machine has to support state transition diagrams comprised of tens of thousands of states and state transitions, and should be scalable to larger numbers of states and transitions.

Current state-of-the-art schemes have only been able to address a subset of these requirements. This section will present a new programmable state machine that is able to meet all five requirements. This state machine will be denoted as B-FSM, which stands for BART-based Finite State Machine, as will be explained in the following sections.

B. Transition Rules

The B-FSM technology will be described using the example of a simple state transition diagram shown in Fig. 1, which detects the first occurrence of one of two patterns “121h” and “ABh” (‘h’ means hexadecimal notation) in a stream of 4-bit input symbols. Fig. 1 only shows input symbols; output values will be discussed later.

The B-FSM technology describes a state transition diagram using a small number of so-called state-transition rules, which involve match operators for the current state and input symbol values, and a next state value. The transition rules are assigned priorities to resolve situations in which multiple transition rules are matching simultaneously. This will be explained using the following list of state-transition rules that can be derived for the state transition diagram of Fig. 1:

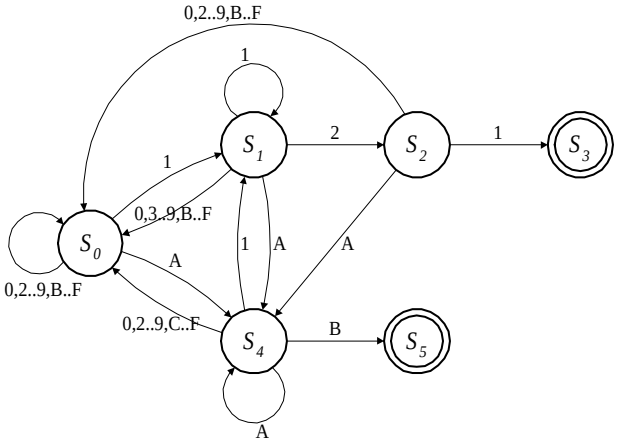


Fig. 1. Example State Transition Diagram.

rule	current state	input	next state	priority
R_1	S_2	1h	S_3	2
R_2	*	1h	S_1	1
R_3	S_1	2h	S_2	1
R_4	S_4	Bh	S_5	1
R_5	*	Ah	S_4	1
R_6	*	*	S_0	0

The '*' symbol (wildcard) represents a "don't care" condition.

The B-FSM operates in the following way. In each cycle, the highest-priority transition rule that matches the current state and input symbol is searched, and is used to determine the next state value. For example, transition rules R_1 and R_2 specify that with an input symbol 1h, a transition will be made to state S_3 if the current state is S_2 and that a transition will be made to state S_1 if the current state is any state other than S_2. This is achieved by assigning rule R_1 a higher priority than rule R_2. Transition rule R_6 can be regarded as a default rule that is used if no other matching rule has been found. As can be verified, the entire state transition diagram in Fig. 1 is described by the six transition rules R_1 to R_6.

Fig. 2 shows a block diagram of the B-FSM based on the operation described. The transition rules are stored in a so-called transition-rule memory, encoded as shown in Fig. 3. The so-called *test part* of each transition rule contains a state field, an input field and a 2-bit flags field indicating whether the state and input are "don't care". The *result part* contains a next-state field and an output field. In each cycle a rule selector will select the highest-priority transition rule that matches the current state (stored in the state register) and input symbol. The result part of the transition rule selected will then be used to update the state register and to generate an output value.

Algorithms have been developed that automatically derive state-transition rules from any kind of state transition diagram (a detailed discussion of this topic exceeds the scope of this paper). However, it has been our experience that for several applications including parsing, it is often more intuitive to describe functions directly using state-transition rules involving match conditions, wildcards, and priorities, than using conventional state transition diagrams.

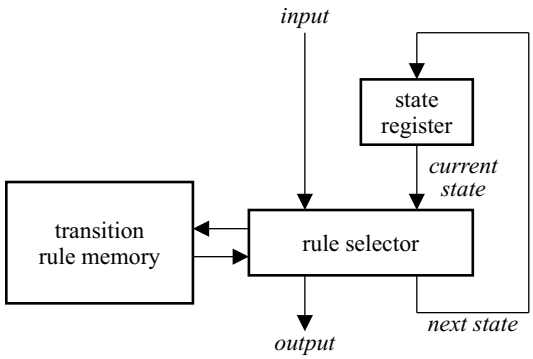


Fig. 2. Rule Selection.

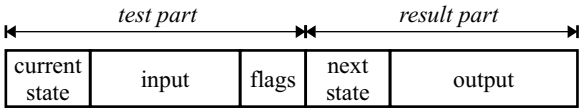


Fig. 3. Transition-Rule Vector.

C. BART-*based Rule Selection*

The rule selector is based on the BART algorithm, which is a scheme for exact-, prefix- and ternary-match searches, supporting fast search performance in combination with high storage-efficiency and fast incremental updates. BART is based on a novel hash function with the special property that the maximum number of collisions for any hash index can be limited by a configurable bound P. The hash index is extracted from bit positions within the search key vector, which are selected to realize the maximum collision bound P. The value of P is based on the memory access granularity to ensure that all collisions for a given hash index can be resolved by a single memory access and by at most P parallel comparisons. For a detailed description of BART, including the fast incremental update function, the reader is referred to [4] [5].

The application of BART for transition-rule selection will now be illustrated using the transition rules described above. The search key in this case consists of the state and input fields in the test part of a transition rule.

The above transition rules listed in binary notation are:

rule	current state	input	next state	priority
R_1	010b	0001b	011b	2
R_2	xxxb	0001b	001b	1
R_3	001b	0010b	010b	1
R_4	100b	1011b	101b	1
R_5	xxxb	1010b	100b	1
R_6	xxxb	xxxxb	000b	0

The 'x' symbol represents a "don't care" bit.

In this example, a 3-bit state vector is used, and state S_0 is encoded as 000b, state S_1 as 001b, and so on.

The underscored bit position, which is the most significant bit of the input symbol, is an example of a hash index for which the maximum number of collisions is limited to $P = 4$. This can easily be verified: If the most significant input bit equalled 0b, then only rules R_1, R_2, R_3, and R_6 could be matching. If this bit were '1', then only rules R_4, R_5 and R_6 could match. In both cases, the number of collisions per hash index value does not ex-

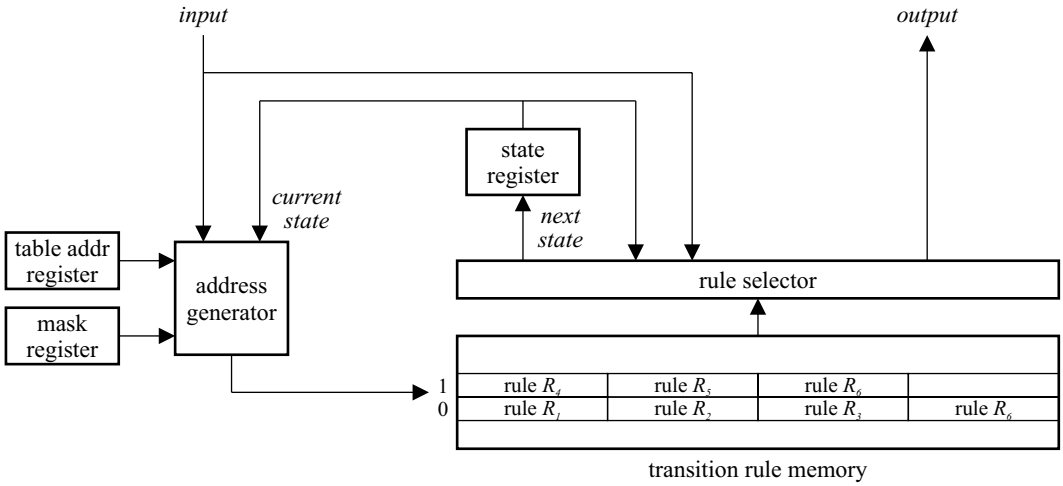


Fig. 4. Transition-Rule Table.

ceed $P = 4$ rules. Note that the hash index is typically extracted from multiple bit positions that are not necessarily adjacent (see [4]).

Fig. 4 shows the corresponding hash table, which will be called transition-rule table. The address generator will extract the hash index from bit positions within the state and input vectors that are selected by a mask stored in the mask register. This index value will be added to the start address of the transition-rule table, which is stored in the table-address register, to obtain a memory address. The transition-rule table entry that is read from the transition-rule memory contains at most P transition-rule vectors corresponding to the transition rules mapped on the hash index. These transition rules are ordered by decreasing priority within each table entry. The test parts of these P transition rules are compared in parallel with the actual values of the state register and input symbol, while taking potential "don't care" conditions indicated by the flags field into account. The result part of the first, and therefore highest-priority, matching transition rule, will be used to update the state register and to generate an output value.

D. State Clusters

To support very large state transition diagrams involving multiple thousands of states and transitions, the state transition diagram is partitioned into so-called state clusters. States within a state cluster are regarded as *local* states that are assigned state vectors that are only unique within the same state cluster. As a result, a state can only be identified uniquely by the state cluster in which it is located in combination with its local state vector. This concept is illustrated in Fig. 5, which shows an example of the partitioning of the (small) state transition diagram of Fig. 1 into two state clusters, involving local states that are only unique within the same state cluster. Note that state clusters will typically contain many more states and transitions than are shown in Fig. 5.

For each state cluster, a transition-rule table is created using the BART algorithm as described in Section II-C and stored in the transition-rule memory shown in Fig. 4. State transitions within the same state cluster are encoded using transition-rule vectors as shown in Fig. 3 and are executed as described in Section II-C. State transitions to different state clusters are encoded using the transition-rule vector shown in Fig. 6, which includes an additional table address and mask field. The values of these fields are used to update the corresponding registers in Fig. 4 in order to realize a "transition" to a different transition-rule table that "implements" the state cluster in which the next state is located. The type of a transition-rule vector (Fig. 3 or Fig. 6) is encoded using a flag bit (not shown).

The rationale behind the state cluster concept is the following. The combination of the table address and a (local) state value can be regarded as a global state vector that uniquely identifies each state in the entire state transition diagram. Now only a small part of this vector (the local state) is actually processed; namely, bit extraction to form an hash index value, and parallel testing against the state fields of at most P transition-rule vectors. The larger part of the global state vector (the table address) is directly used to generate a memory address. As a consequence, the B-FSM concept can be scaled to state transition diagrams consisting of tens of thousands of states and transitions, and even beyond, by simply enlarging the width of the table address vector, without increasing the processing complexity. The only performance impact would arise from the access and cycle times of a larger memory.

A second advantage of the state-cluster concept is that it allows both the storage-efficiency and update performance to be improved because the BART algorithm is applied on smaller portions of the data structure. Algorithms are being developed for automatic partitioning of a state transition diagram into state clusters; however, a detailed discussion of this topic would exceed the scope of this paper.

E. Performance Evaluation

The "execution" of a state transition by the B-FSM involves three phases: (1) address generation, (2) memory access, and (3) transition-rule selection, including state-register update and output generation. Several mechanisms have been developed to speed up each of these phases. For example, the implementation of the address-generator function can be simplified significantly through state encoding optimizations, proper alignment of the transition-rule tables, and by exploiting characteristics of

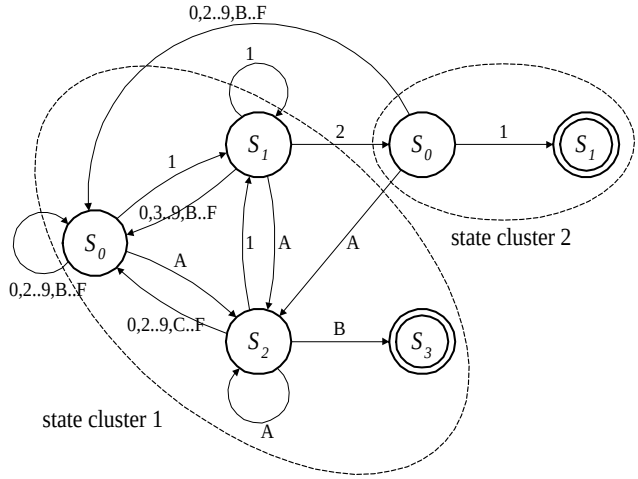


Fig. 5. State Clusters.

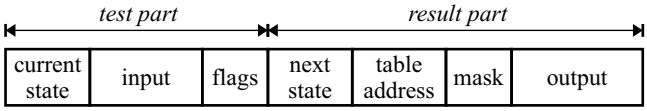


Fig. 6. Rule Vector for Transitions to a different State Cluster.

actual state transition diagrams for the intended application (in this case, XML processing).

The feasibility and the performance of the B-FSM concept including these optimization mechanisms have been validated using simulation models and an FPGA-based hardware prototype. A detailed discussion and performance analysis, however, will be postponed to a future paper owing to space limitations and ongoing further evaluation work. Instead, the following statements are made regarding the design objectives stated in Section II-A:

1. *High performance*: Synthesis experiments have shown that an execution rate of one transition per clock cycle should easily be achieved for clock frequencies of several hundred MHz. Ongoing experiments investigate the maximum frequency, which we expect to be in the GHz range.
2. *Storage efficiency*: The B-FSM is very storage efficient owing to the use of the BART algorithm. In contrast to previous programmable state-machine concepts that use the entire state and input vectors to index a memory, no exponential relation exists between the size of the state and input vectors and the storage requirements. We expect the storage requirements to grow close to linear with the number of transition-rule vectors for several parsing applications.
3. *Programmable*: The B-FSM is fully programmable. The BART scheme supports fast incremental updates, allowing dynamic addition and removal of states and transitions (see [4]).
4. *Wide input and output vectors*: Because the input symbol is only involved in relatively simple processing steps (also a benefit of the optimizations referred to in the preceding section), the input symbol width will not be a performance-limiting factor for symbol widths in the range of 16 to 32 bits. Because output generation is not part of the critical path, the output field in a transition rule can be made as large as desired, and will only impact the required width of the transition-rule memory.
5. *Scalability*: The concept of state clusters allows the B-FSM to support state transition diagrams comprised of tens of thousands of states and state transitions.

III. Software Limitations for XML Processing

Conventional general-purpose processors are characterized by the sequential nature of the instruction execution, in which instructions are selected based on their location (address) within memory using a program counter that is incremented each time an instruction has been executed. Conditions are typically evaluated one at a time, and affect the instruction execution flow by means of explicit conditional branch instructions. For this mode of operation, several mechanisms (e.g., multiple instruction issue, prefetching, branch prediction) have been developed to optimize the performance [6]. Because of the nature of these mechanisms, state-of-the-art processors will be able to achieve good performance for applications involving a predictable instruction execution flow in which conditional branches and other less predictable events form a relatively small portion of the total instruction count.

A key function of an XML parser consists of the evaluation of multiple conditions that can occur at the granularity of strings and even at individual characters in the XML document being processed. Examples of such conditions are testing whether a character is a legal name character, whether an end tag matches a previously processed start tag and is correctly nested, whether an attribute name is unique for a given element, and so on. The nature and frequency at which these conditions occur result in a less predictable instruction flow, and consequently, in a non-optimal performance on a general-purpose processor as described above (e.g., owing to execution-pipeline stalls in the case of mispredicted branches). A related problem is that conventional processors do not provide an efficient way of evaluating multiple conditions of various types in parallel.

Other (fundamental) performance bottlenecks arise from the limited amount of parallelism available on a conventional processor, which does not allow the streaming nature of XML parsing to be exploited efficiently by processing the XML document in a pipelined fashion, and from the inability to process variable-length encoded character streams efficiently (e.g., UTF-8).

IV. XML Accelerator

This section provides an introductory description on how the B-FSM technology presented in Section II is used as core technology for a novel XML Accelerator Engine called ZUXA (Zurich XML Accelerator), which is intended to overcome the software bottlenecks discussed in Section III.

ZUXA consists of two main components that are shown in Fig. 7: the programmable state machine (B-FSM) and an instruction handler. The former dispatches instructions to the latter, which are selected based on state information, the input character stream and processing results received from the instruction handler. The instruction handler implements the ZUXA "instruction set", which includes a variety of string and character processing functions that can be applied to the input character stream and used to generate output.

The following sections will describe the high-level operation of the programmable state machine and instruction handler. This

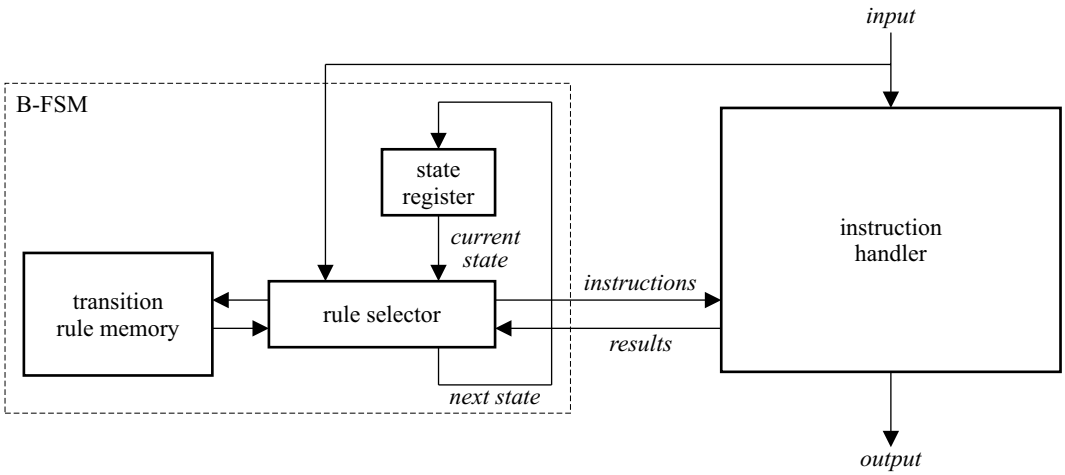


Fig. 7. The ZUXA XML Accelerator.

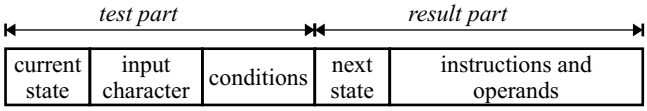


Fig. 8. ZUXA Transition-Rule Vector.

description will be restricted to a few functions and is only intended to illustrate the basic concept. A future paper will discuss the ZUXA concept and functionality in more detail.

A. Programmable State Machine

The "input" to the programmable state machine consists of the input character retrieved from the input stream and the processing results received from the input handler. The "output" consists of instructions and operand values that are dispatched to the input handler. Fig. 8 shows the corresponding ZUXA transition-rule vector, in which the original input field (Fig. 3) is replaced by an input character field and a conditions field, whereas the original output field is replaced by an instructions and operands field. Because the B-FSM supports wide input vectors, see Section II, transitions can be made based on a set of multiple conditions (e.g., eight) in combination with wide input character vectors, for example, 16-bit UTF-16 code units or even wider. The wide output vector supported by the B-FSM allows multiple instructions and operand values to be associated and dispatched with each state transition.

The ZUXA concept can be regarded as a processing model that allows a very flexible "programming" of a large number of potential execution paths in the form of an enhanced state transition diagram, in which the actual path through the diagram taken during the program execution is selected by real-time evaluation of different sets of multiple conditions for each transition. In a more general form, the ZUXA concept can be seen as a processing model in which instructions are associated with sets of multiple conditions (including wildcards and priorities). In each "execution cycle", all these conditions are evaluated and the instructions for which all conditions evaluate positively are selected for execution. There is some similarity of this approach with a CAM (content addressable memory), where a memory location is selected based on its contents ("conditions"), whereas a conventional processor architecture can be compared with an SRAM, which selects a memory location only based on its address.

B. Instruction Handler

This section will focus on the basic concept of interaction between the programmable state machine and the instruction handler. It will be restricted to a few functions for illustration purposes. A complete description will be postponed to a future paper, as mentioned above.

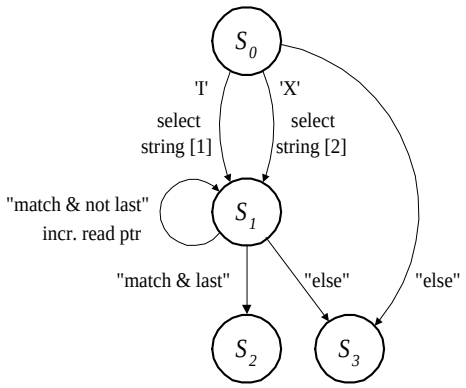
Because of the high performance of the programmable state machine and its ability to process large input vectors, it can react quickly to a large number of input events, in this case the input character stream and processing results provided by the instruction handler. The instruction handler, therefore, does not have to implement functions that have to execute for a longer period in stand-alone fashion, but can instead implement simpler (and faster) functions that run under tight control of the programmable state machine.

This concept will now be illustrated using an example in which the input character stream is tested against two strings, "Internet" and "Xml parser". Figs. 9 (a) and (b) show five state-transition rules and the corresponding state transition diagram, which comprise the "program" executed by the programmable state machine. The initial state is state S_0. If the current input symbol is 'I' then transition rule R_1 will match, involving a transition to state S_1 and an instruction "select string[1]" will be dispatched to the instruction handler. This instruction will position a read pointer at the first character of string 1 in the character memory shown in Fig. 9 (c). If the current input symbol is 'X' then transition rule R_2 will match and the read pointer will be positioned at the start of string 2.

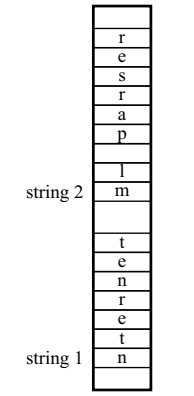
Rules R_3 and R_4 involve two conditions denoted as "match" and "last", which the instruction handler provides to the programmable state machine as part of the results vector shown in Fig. 7. The "match" condition evaluates to true if the current input symbol matches the "current" character in the selected string, i.e., the character referred to by the read pointer. The "last" condition evaluates to true if the read pointer refers to the last character of the selected string in the character mem-

rule	current state	input	conditions	next state	instructions	priority
R_1	S_0	'I'	*	S_1	select string[1]	1
R_2	S_0	'X'	*	S_1	select string[2]	1
R_3	S_1	*	match, not last	S_1	incr. read ptr	1
R_4	S_1	*	match, last	S_2	nop	1
R_5	*	*	*	S_3	nop	0

(a)



(b)



character memory

(c)

Fig. 9. String Compare.

ory. Rules R_3 and R_4 implement a string-compare function that tests a selected string in the character memory against the input character stream. Each time the current input character matches the current character in the selected string, rule R_3 is executed, involving a transition to (the same) state S_1 and an increment of the read pointer by one. Rule R_3 is iterated until the last character in the selected string has been reached, which, in the case of a match, will trigger transition rule R_4 involving a transition to state S_2. State S_2 will therefore be reached if the input character stream matches with either string 1 or string 2. If the input character stream does not match with one of these two strings, then a transition to state S_3 will be made, based on the "default" transition rule R_5.

This example illustrates the two ways in which the programmable state machine can be used in the parsing process: (1) rules R_1 and R_2 parse the input character stream directly by immediate testing of the input characters, and (2) rules R_3 and R_4 control the instruction handler to perform the actual parsing of the input character stream. The latter mode of operation is very storage-efficient: the number of transition rules and, consequently, the storage requirements of programmable state machine, are independent of the length of the strings involved. Longer strings (e.g., 1000 characters) would only require more storage in the character memory, which is fully optimized for storing string data in contrast to transition-rule vectors, which are optimized for storing "control" data.

In addition to the relatively simple string-compare function described above, the instruction handler will also implement several character- and string-processing functions for writing character strings from the input into the character memory, and for encoding, conversion, searching, filtering, and output generation. Furthermore, it will also provide various computational capabilities as required by several state-of-the-art parsing applications.

C. Performance Evaluation

The ZUXA concept is being validated in two ways: (1) a software-based simulation model and (2) a hardware prototype based on an FPGA [7]. Several programs (state transition diagrams) have been created implementing subsets of the functionality required for a range of XML parsing and processing applications, such as well-formedness checking, validation, canonicalization and transformation (e.g., XML to HTML). These programs are being used to process a large set of actual XML documents, using both the simulation model and the hardware prototype.

Early results prove the feasibility of the concept and have confirmed its potential to achieve a processing rate of one character per clock cycle. Ongoing work includes expanding the functionality of ZUXA to support a wider variety of character- and string-processing functions and optimization of the state transition diagrams based on actual XML document characteristics.

Again, a detailed discussion and performance analysis of the ZUXA concept will be postponed to a future paper.

V. CONCLUSIONS

A novel programmable state machine technology, called B-FSM, has been presented, that provides full programmability in combination with a processing rate of one transition per clock cycle, high storage-efficiency, support of wide input and output vectors, and scalability to tens of thousands of states and state-transition rules.

Building on the B-FSM technology, the high-level concept of an XML acceleration engine called ZUXA has been introduced. ZUXA is intended to overcome performance bottlenecks of software-based XML processing by providing a processing model that is optimized for conditional execution involving large numbers of various conditions, in combination with dedicated instructions for character- and string-processing functions.

REFERENCES

[1] Extensible Markup Language (XML), http://www.w3.org/XML.

[2] The Cover Pages: XML applications, http://xml.coverpages.org/xmlApplications.html.

[3] M. Nicola and J. John, "XML parsing: a threat to database performance," *Proc. of the 12th Intl. Conference on Information and knowledge management*, pp. 175-178, November 2003.

[4] J. van Lunteren, "Searching very large routing tables in wide embedded memory," *Proc. IEEE Globecom*, vol. 3, pp. 1615-1619, November 2001.

[5] J. van Lunteren and A.P.J. Engbersen, "Fast and scalable packet classification," *IEEE Journal of Selected Areas in Communications*, vol. 21, no. 4, pp. 560-571, May 2003.

[6] J.L. Hennessy and D.A. Patterson, "Computer architecture: a quantitative approach,", 2nd edition. Morgan Kaufmann Publishers, Inc., 1996. ISBN 1-55860-329-8.

[7] FPGA-based rapid-prototyping platform Spyder-Virtex-X2/XCV2000, http://www.x2e.de.