Real-Time Capable Hardware-based Parser for Efficient XML Interchange

Vlado Altmann, Jan Skodzik, Peter Danielis, Nam Pham Van, Frank Golatowski, and Dirk Timmermann
University of Rostock

Institute of Applied Microelectronics and Computer Engineering

18051 Rostock, Germany

Email: vlado.altmann@uni-rostock.de

***Abstract*—The use of Extensible Markup Language (XML) for data interchange became standard in most computer systems. The benefits of XML documents are their easy handling, dynamic adjustment to nearly all needs, availability of parsers for any programming language, and the human readability. Despite many advantages of XML, it is still primarily applied in software-based systems. The serious drawback of XML is the size of the document and the text-based parsing. Further, the dissemination of the Web service technology in automation demands very fast data processing especially in real-time scenarios. This implies the compactness of the data representation and fast parsing possibility. In such cases, hardware-based solutions are often used to speed up the process. Efficient XML Interchange (EXI) was developed to leverage the deployment of XML in deeply embedded systems. EXI provides very high XML compression rates without additional processing load. However, EXI parsing was only tested in software-based systems. As very fast processing speed is targeted in this work for, e.g., real-time systems, which software-based parsers most probably cannot achieve, we provide an investigation and the prototype implementation of the EXI parser in hardware and compare it to software-based solutions.**

Keywords—Web services; Hardware; Communication networks; Embedded software; Real time systems

I. INTRODUCTION

In recent years, the significant dissemination of Web service technology led to new applications in nearly all sectors. Especially in home, building, and factory automation, Web services are enjoying great popularity [1]-[3]. The installations benefit from standardized, self-describing, and adjustable interfaces. Specific profiles were developed to deploy this technology even in deeply embedded systems. A representational example is Devices Profile for Web Services (DPWS) [4]. DPWS comprises a minimum set of Web services standards required to find, describe, control, monitor, and represent any technical device. The use of DPWS in automation scenarios, e.g., smart metering was already suggested. The implementation of embedded Web services reaches from enterprise systems to sensor nodes. Thus, the universal character of Web services is approved. DPWS or embedded Web services use XML for data representation and HTTP for data transport. A specific parser is required to extract the data from the XML document. XML parsers are available for nearly all programming languages since XML is a common data exchange format. The parsing of XML documents can be basically considered as string parsing, which is software friendly.

The main drawback of the XML format is its size. Depending on the document structure, the overhead can be several times bigger than the payload. In some automation scenarios such as a real-time systems or deeply embedded devices, this can become critical. In order to benefit from the Web service technology in such systems, Efficient XML Interchange (EXI) was developed [5].

In contrast to common compression methods, EXI uses the knowledge of the XML format and if necessary XML structure. This allows very high compression rates of up to 90 % [6][7]. EXI represents XML tags as a sequence of events. Events can be encoded with only few bits depending on the EXI mode. Another important difference of EXI from common compression methods is that data can be directly written into or read out from the EXI document without later compression or prior decompression. Thus, no additional CPU and memory resources are required for computation.

Nevertheless, in order to extract data from the EXI document an appropriate parser is required. Some EXI software parsers were already introduced. However, we focus our work on the very fast data processing in, e.g, real-time systems. In such scenarios, processing speeds of some µs are required [8]. In this work, we investigated the possibility of the implementation of an EXI parser in hardware for very high processing speeds. In other scenarios, the suggested EXI parser can be used as digital signal processor (DSP) to relieve the main CPU. In our work, we focus on the dynamic characteristic of the hardware EXI parser making it adjustable to any application as well as the use of standard interface buses making it integrable in other systems.

Briefly summarized the main contributions of this paper are the following:

- Investigations of the requirements for realizing an EXI parser in hardware

- Design and implementation of a hardware EXI parser
- Evaluation of the hardware/software co-design
- Comparison with existing solutions

The remainder of this paper is structured as follows. Section II gives a short overview about related work in the field of EXI. Section III describes the principles of EXI. The hardware EXI parser is presented in Section IV. The hardware/software co-design is described in Section V. After the evaluation of the suggested solution in Section VI, the paper concludes in Section VII.

II. Related Work

The actual final version of EXI 1.0 was finalized in 2011 by the World Wide Web Consortium (W3C) [5]. The comparison of compression rates between EXI and GNU zip (gzip) was carried out in [6]. The evaluation has shown that EXI produces much smaller streams than gzip. The benefit of EXI in collaboration with DPWS and Constrained Application Protocol (CoAP) was already approved in [9] and [10]. Since EXI is a relatively new compression method, only some implementations are known. The most full-featured EXI implementation is EXIficient [11]. It is programmed in Java and can encode EXI documents from XML and vice versa. Many EXI options are supported. Another Java implementation of EXI is OpenEXI [12]. Both of these implementations expect powerful hardware and an operating system that supports a Java virtual machine. The next two implementations uEXI [13] and EXIP [14] are both written in C and intended to be used in embedded devices. The processing speed is dependent on the hardware they run on and can highly deviate. In order to evaluate the suggested hardware-based solution, we compare the performance with software implementations using a Zynq Z-7020 chip, which consists of Dual Core ARM Cortex-A9 and Artix 7 Field Programmable Gate Array (FPGA). Moreover, a hardware/software co-design is evaluated as well.

III. EXI Basics

EXI is a binary XML encoding. The XML tags are represented by events. An EXI document consists of a sequence of events. XML attributes can be described with the aid of additional context, which can be assigned to events. In contrast to XML, the element values are encoded according to their data type. For instance, an integer is encoded as a 4 byte value instead of 10 byte string. EXI provides different modes in order to provide an adjustable tradeoff between compression rate and lossless XML compression. For example, XML namespace prefixes can be preserved and are not replaced by IDs. This results in an exact XML reconstruction but also in a larger document size. EXI uses grammars for encoding. With the help of grammars, event codes are generated. Events with a higher probability of occurrence are encoded with fewer bits.

EXI differentiates between two modes: schema less and schema informed. Schema less mode is used if no information available about the XML structure. In this case, built-in grammars are used for event code generation. These grammars are continually extended with new context while parsing the XML document. By this learning mechanism, equal strings must be encoded only once. Better compression rates can be reached with schema informed mode. If the structure of the XML document is known by means of XML Schema Definition (XSD) then all possible events are also known in advance. In this case, the generated event codes can be much shorter. Schema informed mode differentiates between strict and non-strict mode. In the strict mode, the XML document cannot deviate from the XML Schema. In a non-strict mode, the deviation is allowed. In this case, learning built-in grammars are used for unknown events. However, non-strict mode results in longer event codes and thus in larger document size. Since schema-informed strict mode provides the best possible compression rate and communication data structures are known in embedded systems, this mode is considered in this work.

IV. Hardware EXI Parser

The goal of the EXI parser is to extract the data from an EXI stream. On the one hand, in order to make an EXI parser usable for any application and any XML Schema, it must be generic. On the other hand, a generic parser in runtime would result in very high hardware resource consumption. Thus, in order to reach the most efficient implementation and save hardware resources, we suggest the development of a generic EXI parser at synthesis time. In order to synthesize the hardware, Very High Speed Integrated Circuit Hardware Description Language (VHDL) is used. The VHDL description of the EXI parser must be dynamically generated depending on the XML Schema. XML Schema can be provided by the device manufacturer or defined in the specification for some specific service type.

For code generation and parsing of XML Schema, the open source software EXIficient is used. EXIficient is the Java implementation of W3C EXI specification. It is able to convert XML files using XML Schema to EXI streams and vice versa. EXIficient XML Schema parser and grammar builder is used to leverage VHDL code generation. EXIficient was extended with additional classes that use the internal representation of EXI structure to generate VHDL code. Then, the VHDL is used for hardware synthesis. The steps for EXI parser generation are depicted in Figure 1.

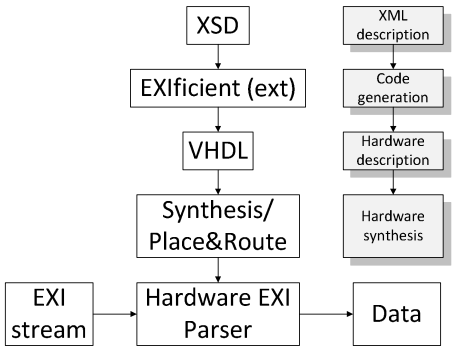

Figure 1: Hardware EXI parser generation

A. Hardware description

For better comprehension of the VHDL code generation, we want to discuss the final structure of the VHDL code before the synthesis first. As previously mentioned, EXI uses events for XML structure mapping. The sequence of events is represented by a finite-state machine (FSM). FSMs can be easily implemented in hardware making EXI hardware friendly in contrast to XML. Depending on the event code, the FSM has to change to the next state. For better comprehension, let us consider the following simplified example (see Figure 2). The XML snippet has one upper element and 4 sub-elements. After the upper element is reached, 4 different events (elements) can occur. We need only 2 bits to encode each event for sub-elements. Element 0 can be binary encoded as 00 and element 3 as 11.

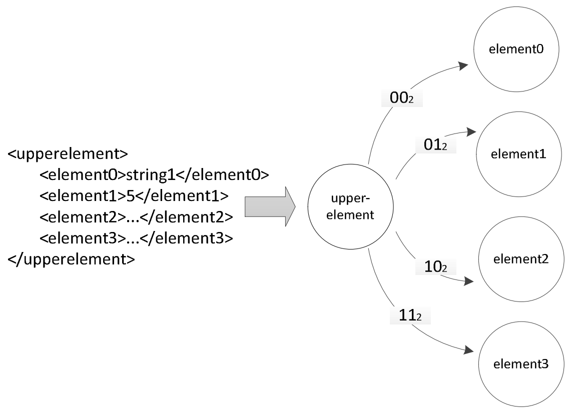

Figure 2: Structure parsing

Depending on the bit sequence in the EXI stream, the FSM will go to the according state. By this means, the hardware EXI parser can go through the EXI document. The states are generated according to the XML Schema and are used for structure parsing.

In addition to the structure parsing, data parsing is the second essential part of EXI. As previously mentioned, EXI encodes data according to its type. For instance, an integer value is encoded as compound of a Boolean and an unsigned integer. The Boolean part is encoded with one bit and denotes the sign of the integer value. The unsigned integer is encoded as a sequence of integer bytes and can be of any length. The most significant bit of every integer byte indicates the presence of the next byte. By this means, the sequence can be terminated. In order, to extract the final value, the 7 least significant bits of every byte must be concatenated in the little endian order.

As previously mentioned, every string in EXI is encoded only once. The strings are assigned to QNames. A QName consists of a Uniform Resource Identifier (URI) and a LocalName. QNames represent XML tags. For the purpose of string parsing, the parser must keep two tables. The first one is a global string table and the second one is a local string table. A so-called *local hit* implies that the same string was already encoded for this QName. The index of it must then be looked up in the local string table. A *global hit* implies that the same string was already encoded for some other QName and the index must be looked up in a global string table. If the string has a hit neither in the local nor in the global table, it will be considered as new and injected into both tables. The indexes of both tables will be incremented. The number of bits for index representation is not static and is determined depending on number of entries in the tables.

In order to save hardware resources, data parsing is not hardcoded into every path of the main FSM but swapped into special data parsing states. Then, the hardware resources for every data type exist only once. However, the logic to control the choice of the next state becomes more complex. The next state of the main FSM must be saved and later restored after data parsing is done. For every data type an own parsing routine is generated.

The parsing procedure is depicted in Figure 3. SE denotes the *Start Element* and EE the *End Element* event. The element values are described by CH (characters), since all values in XML are represented by strings. EXI parser can also deal with attributes and their values, which are not shown in the example for reasons of clarity.

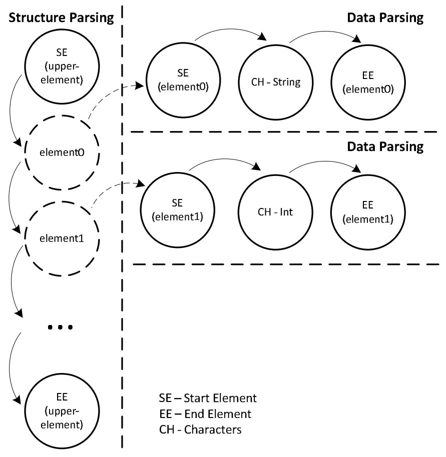

Figure 3: Data parsing

In order to communicate with other components, the EXI module needs interfaces. Apart from the clock and reset pins, the EXI parser provide a 32 bit input bus, input valid, ready and read out signal. If the module is signaled as ready an input can be fed in and an input valid signal set. An input bus expects an EXI stream divided in 32 bit words. The first 32 bit word must indicate the length of the stream. As output serve 32 bit output bus, output valid, processing done and an error flags. The processing done flag indicates the successfully completed end of processing of one EXI stream and an error flag indicates a malformed EXI stream. In order to read the data from the EXI parser, the read out flag has to be set. The output bus represents the payload data transported with the EXI stream with metadata prepended. The data on the bus is valid if the output valid flag is set. The data is identified by an ID, so that other modules can properly assign it. The IDs are created during the EXI parser generation.

The whole EXI hardware parser is generated in two process methodology making it easy to understand and debug. In the prototype implementation, the incoming data are stored in a First In First Out (FIFO) memory, which consist of registers. After reset, the system is in idle state. This state is also used internally to reset used registers and clean the memory after processing of an EXI stream. The FSM goes then into an EXI initialization state and waits for input. When the input valid bit is set, the system writes the incoming bit stream into the mentioned input FIFO. After the stream was completely written, the FSM changes to the EXI header processing state. In this state EXI distinguishing bits (DB = "10") and presence bit for EXI options (PBEO) are checked. Since this prototype implementation does not support any EXI options, the presence bit must be 0. If this check is passed, the FSM changes to EXI body processing (see Figure 2 and Figure 3). The processing of the EXI body takes place as described above. If an error in the EXI header or body occurred, the processing is interrupted. This is accompanied by setting the error flag and jumping into the idle state of the FSM. If the finish state is reached, the processing was successful and the processing done flag is set.

In order to store the next state, while processing the data, a stack memory is used. The stack is implemented by means of registers. Before changing into the processing state of some special data type, the next structure state is pushed on the stack. Then, data type processing is performed. After the processing is done, the next state is loaded from the stack and the main FSM can proceed with the structure parsing. The extracted payload data must be cached in the parser and forwarded after the processing of the complete EXI stream is done. For this purpose, a FIFO is generated for every data type. Alongside with the values, an additional metadata is stored. This metadata can be used later by other modules to assign the values to correct elements.

Data output can be initiated by any connected module by setting the *read out* bit. In the first data output state, the stored meta-information is evaluated. Depending on the type some conversions can be done. In the second state 32 bit metadata is created and sent to the output. Thereby, the *output valid* flag is set. In the next state, the payload data is read out from the FIFO und forwarded to the output. The procedure is repeated until the FIFO is empty. Afterwards, the next state is taken from the stack and the output processing continues. All output states start the value output with 32 bit metadata. This metadata comprises data type, LocalName ID and URI ID. This information helps other modules to properly assign the values to elements. Data representation after the metadata depends on the current data type. The output can contain many read cycles, e.g., in case of strings.

B. Code generation

As described above, the EXI parser is generated at synthesis time. VHDL code generation can be subdivided into two parts: a static and a dynamic description. The static description comprises entity, signals, components, processes, and variables declaration und is independent from XSD. Moreover, the static description provides FSM states for data parsing. The dynamic description primarily comprises FSM states for structure parsing. The result of the code generation is an *exi_parser.vhd* file.

The suggested code generator utilizes EXIficient classes for EXI grammar creation. EXIficient subdivides events into fragments of grouped elements. This structure is converted by the suggested code generator into an array of the class *State,* which will be used for structure states creation. This class consists of the following fields: event name, event type, attribute name, data type, namespace URI, URI ID, number of branches, and branch array. With the help of the *State* class, the structure of XML can be mapped on the *State* array. Every element starts with the Start Element (SE) event type. All information related to this event is entered into the first field of the *States* array. After SE, any other start elements (SE), attributes (AT) or characters (CH) can follow. The *States* array is filled up until all branches of one fragment are traversed. While adding a new entry into the array, the generator checks whether this entry is already present. In this case, no new entry is created but the index of the known state is entered into the branch array of the last event. The processing of one fragment is accomplished if End Element (EE) event type is reached.

After all events of one fragment have been processed, a function for code generation is called. This function creates state names and state transitions for FSM. For this purpose all entries of the *States* array are traversed. For the first entry in the *States* array one state for the FSM is generated. All other entries represent state transitions. Before the next state is created, the generator checks the event type. When the next event is of type CH or AT with any data type, VHDL code for transition into an according data type processing state is generated. Moreover, the next state must be stored in order to return back to structure processing. Additionally, the corresponding URI ID and LocalName are stored for data classification. When the next state is neither CH or AT a simple state transition is added. By reaching the event type EE the processing for the current fragment is done. Then, the processing of the next fragment begins. The described processing of element fragments is presented in a simplified way in Table 1.

TABLE 1: FSM STATES CREATION BY MEANS OF ELEMENT FRAGMENTS

States Array Index	Fragment 1	Fragment 2	Fragment 3
0	SE – element0	SE – element1	SE – upper-element
1	CH (String)	CH (Integer)	SE – element0
2	EE – element0	EE – element1	SE – element1
4	-	-	EE – upper-element

For the state generation of the global elements, the number of such elements is important. It determines the number of bits that must be processed from the EXI stream in the first step. In every step, the generator checks the existence of branches. Therefore, the branch value of the *State* class is examined. If the value is greater than one, a jump into different states is possible. In order to decide which state should be the next, the right number of bits from the EXI stream must be processed. This depends on the number of branches and can be determined by examining the eponymous *State* parameter.

V. HARDWARE/SOFTWARE CO-DESIGN

In some embedded systems the hardware undertakes some time critical or heavy computation tasks. The processed data is then forwarded to a software component for further processing. In this work, we also want to evaluate the possibility of the deployment of the suggested hardware EXI parser as a part of the hardware/software co-design.

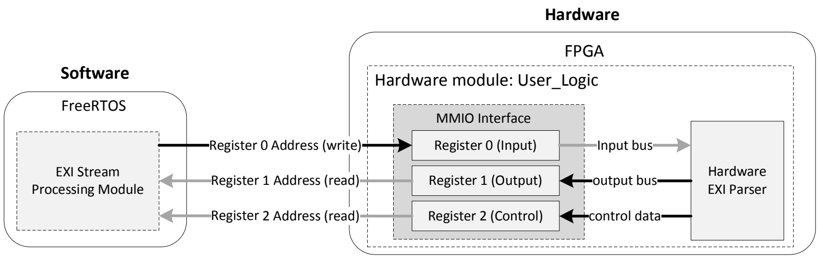

Figure 4: Hardware/software co-design

In order to make the software component easy to use, its creation can also be performed by the suggested EXI parser generator. In the prototype implementation, we are using C++ programming language for the software component. It consists of two files ExiStreamProcessing.h and ExiStreamProcessing.cpp as common for C++. Both files are generated alongside with the VHDL code. However, their use is optional and intended for the hardware/software co-design. In the same manner like the hardware part, the software comprises statically and dynamically generated code instructions. Libraries import or global definitions are examples for static code fragments. For a clear assignment of EXI payload data and easy use of the software, variable names are generated with the corresponding LocalNames of XML elements. After EXI stream processing and forwarding to software, this variables can be used to access the payload data.

An important software function is the evaluation of the data coming from the hardware. The data is assigned to the variables based on LocalName and URI ID. This functionality is realized with the help of a lookup table, whereby URI IDs points to LocalNames. Moreover, code instructions are added to evaluate and store the data type in an appropriate variable. Thereby, the EXI data types determine the type of the variables. For data interpretation, special bit masks are defined. With their help, the parameters can be quickly recognized and assigned to the proper variables. The functions for processing of specific data types can be generated statically. In order to access XML element values, C++ structures are used. The access to element values occur with the following format: *NamespaceName.LocalName*. Both this parameters can be found in the XML document and easily assigned.

In the suggested prototype, the software not only reads out the data but also feeds the hardware EXI parser with an EXI stream. For easy communication between the software and hardware components, Advanced Micro Controller Bus Architecture Advanced eXtensible Interface (AXI) is utilized. In particular, AXI4-Lite is used. AXI4-Lite is a subtype of the AXI interface, which is designed for simple communication. AXI4-Lite is a 32 bit bus. Three additional 32 bit registers are required for Multiple Input Multiple Output (MIMO) interface. The hardware/software co-design is depicted in Figure 4 for better comprehension. Register 0 and 1 are used for input and output respectively. Register 2 is used for control information such as output valid, processing done, and error flags. The remaining bits are reserved. The bus access control is done automatically over the AXI interface so that the modules connected to the bus only need to provide the AXI User Logic as shown in Figure 4. Register of the MIMO interface utilizes the principle of memory mapped input/output (IO). Thus, in order to communicate with the hardware EXI parser the software should only access the corresponding registers over special memory addresses.

VI. Evaluation

In order to evaluate the suggested hardware-based EXI solution, an evaluation platform ZedBoard is used. A ZedBoard comprises a Zynq Z-7020 chip, which is a single chip solution for FPGA and ARM CPU applications. For a precise timing evaluation as well as comparison to other solutions, the real-time operation system FreeRTOS is used. The structure of the suggested solution is shown in Figure 1 and Figure 4. As input in the form of the XML structure description, XSD is utilized and passed to the extended EXIficient framework. In EXIficient, the required hardware description in VHDL as well as the software component in C++ are generated. Moreover, EXIficient creates an EXI stream, which is used for testing later. The hardware component is then synthesized and transferred to the ZedBoard together with the software. The FPGA system reaches a frequency of 50 MHz. The ARM CPU runs with 667 MHz.

In the first tests, the processing speed of the hardware only approach was tested. For this purpose, different standard Web service messages were used such as Hello, Bye, Probe, ProbeMatch, service invocation, and invocation response. The XSD file was created in such a way that all these messages can be parsed with one parser. The experiment has shown that the processing speed is linearly depending on the size of the EXI stream. The measured results are depicted in Figure 5. The resource consumption of the experimental setup amounts to 12012 registers (12 % of total amount) and 12800 lookup tables (24 % of total amount). The resource consumption depends on the number of different EXI states. In the experimental setup, a full Web services system can be mapped on the suggested solution (all essential message types are supported) and fit into a relatively small FPGA.

In the next experimental setup, we evaluated the hardware/software co-design. For this purpose, the same standard Web service messages are used. The behavior is also linear. However, the processing speed is much lower due to the limiting software component. Moreover, the AXI interface requires additional FPGA resources, which are for the co-design 12999 registers (12 % of total amount) and 17868 lookup tables (33 % of total amount). Despite slightly higher resource consumption, the co-design still fits into the FPGA.

In order to compare the suggested solution with an existing software solution, we ported uEXI implementation to our evaluation platform. The processing time of the pure software realization can also be found in Figure 5. The direct comparison of processing of some specific message types of all experimental setups is shown in Figure 6. As shown above, the hardware EXI parser consumes about 5 % of processing time compared to the pure software implementation. The hardware/software co-design requires 73 – 82% of processing time compared to the software solution. The speed up becomes obvious with rising EXI stream lengths.

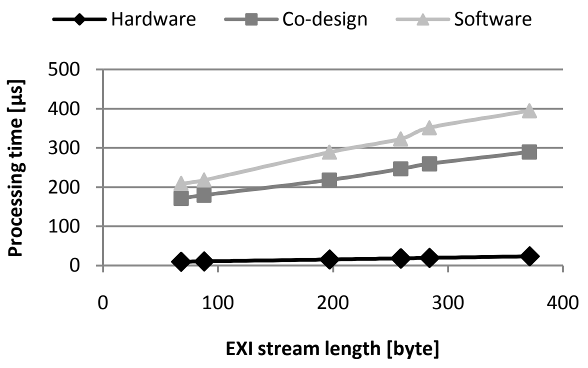

Figure 5: Processing time of different EXI parser implementations depending on EXI stream length

VII. Conclusion

In this work, we provide a prototype design and implementation of a hardware EXI parser. The parser can be dynamically generated at synthesis time and can be adapted to any XML Schema. Thus, the deployment of Web service technology based on XML is possible in deeply embedded or real-time environments. Due to EXI compression, it is possible to parse XML documents in hardware. The suggested EXI parser yields a reduced processing time by about 23 % with a hardware/software co-design and about 95 % as a pure hardware solution compared to software-based approaches. The deployment of a hardware EXI parser can be especially advantageous for hard real-time applications with short response times such as motion control. Thereby, the dynamic structure and interoperability of Web services are not limited. Moreover, EXI parser can be used as a co-processor for EXI parsing and relieve the main processor. By using standard components for the parser generation, it can also be easily integrated into existing FPGA-based systems. The provided software component can receive the data from the hardware parser and forward it to the next processing instance.

In our investigation, we have shown that XML documents can be also used in hardware-based systems with the help of EXI opening new area of deployment for Web services technology.

Acknowledgment

We would like to thank the German Federal Institute for Research on Building, Urban Affairs and Spatial Development (BBSR) within the Federal Office for Building and Regional Planning for their support in this project. This work is partially granted by BBSR within the scope of project "Zukunft Bau".

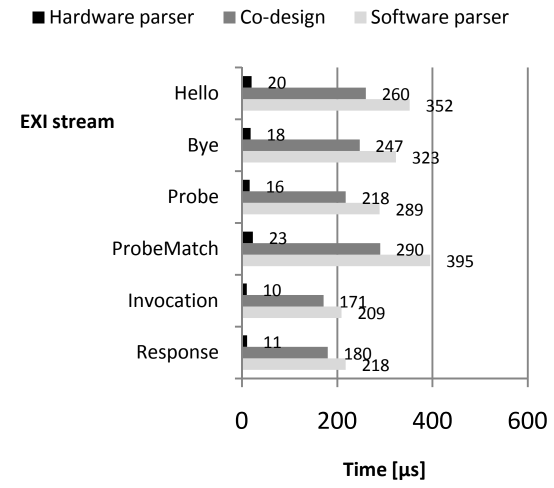

Figure 6: Comparison of processing time of different EXI parser implementations for specific Web service messages

References

[1] M. Kovatsch, M. Weiss, and D. Guinard, "Embedding internet technology for home automation", *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, 2010.

[2] M. Neugschwandtner, G. Neugschwandtner, and W. Kastner, „Web Services in Building Automation: Mapping KNX to oBIX", *5th IEEE International Conference on Industrial Informatics*, 2007.

[3] F. Jammes, B. Bony, P. Nappey, A.W. Colombo, "Technologies for SOA-based distributed large scale process monitoring and control systems", *The 38th Annual Conference on IEEE Industrial Electronics Society (IECON)*, 2012.

[4] D. Driscoll and A. Mensch, "Devices profile for web services version 1.1", OASIS, 2009.

[5] J. Schneider and T. Kamiya, "Efficient XML Interchange (EXI) Format 1.0", World Wide Web Consortium (W3C), 2011. [Online]. Available: http://www.w3.org/TR/exi/.

[6] C. Bournez, "Efficient XML Interchange Evaluation", World Wide Web Consortium (W3C), 2009. [Online]. Available: http://www.w3.org/TR/exi-evaluation.

[7] G. Moritz, D. Timmermann, R. Stoll, and F. Golatowski, "Encoding and Compression for the Devices Profile for Web Services", *IEEE 24th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pp. 514-519, 2010.

[8] P. Neumann, "Communication in industrial automation – what is going on?", *Control Engineering Practice*, pp. 1332-1347, 2007.

[9] V. Altmann, J. Skodzik, F. Golatowski, and D. Timmermann, „ Investigation of the Use of Embedded Web Services in Smart Metering Applications", *The 38th Annual Conference of the IEEE Industrial Electronics Society (IECON)*, pp. 6176-6181, 2012.

[10] A.P. Castellani, M. Gheda, N. Bui, M. Rossi, and M. Zorzi, "Web Services for the Internet of Things through CoAP and EXI", *Communications Workshops (ICC),* 2011

[11] EXIficient. [Online]. Available: http://exificient.sourceforge.net.

[12] D. Dawson and T. Kamiya, "A Quick Introduction to OpenEXI", *The OpenEXI Project*, 2012

[13] C. Lerche, N. Laum, G. Moritz, E. Zeeb, F. Golatowski, and D. Timmermann, „Implementing Powerful Web Services for highly resource-constrained devices", *Pervasive Computing and Communication Workshops (PerCom Workshops)*, pp. 332-335, 2011.

[14] R. Kyusakov, "Efficient XML Interchange Processor", 2013. [Online]. Available: http://exip.sourceforge.net.