Zebra : Building Efficient Network Message Parsers for Embedded Systems

Julien Mercadal, Laurent Réveillère, Yérom-David Bromberg, Bertrand Le Gal, Tegawendé F. Bissyandé, Jigar Solanki

Zebra: Building Efficient Network Message Parsers for Embedded Systems

Julien Mercadal, Laurent Réveillère, Yérom-David Bromberg
LaBRI, University of Bordeaux, France

Bertrand Le Gal
IMS, University of Bordeaux, France

Tegawendé F. Bissyandé, Jigar Solanki
LaBRI, University of Bordeaux, France

***Abstract*—Supporting standard text-based protocols in embedded systems is challenging because of the often limited computational resources that embedded systems provide. To overcome this issue, a promising approach is to build parsers directly in hardware. Unfortunately, developing such parsers is a daunting task for most developers as it is at the crossroads of several areas of expertise, such as low-level network programming, or hardware design. In this paper, we propose Zebra, a generative approach to drastically ease the development of hardware parsers and their use in network applications. To validate our approach, we have used Zebra to generate hardware parsers for widely used protocols, namely HTTP, SMTP, SIP, and RTSP. Our experiments show that Zebra-based parsers are up to 11 times faster than software-based parsers.**

I. Introduction

Embedded systems are increasingly required to interact both among them and with legacy infrastructures to provide advanced services to end-users. This kind of communication among heterogeneous entities requires a protocol to manage their interaction. Traditionally, because of their highly constrained resources, they have used non-standard, application-specific, binary protocols where message parsing and message construction are simple [18]. The use of non-standard protocols, however, complicates the interaction with other systems, as it is required in many emerging applications. Thus, attention is turning to the use of standard text-based protocols. For example, the SIP protocol is now being used in sensor networks [8] and mobile ad-hoc networks [9], [16].

Standard text-based protocol message parsers are typically implemented in software as Finite State Machines (FSM), using a low-level language such as C to provide efficiency. However, developing such parsers is challenging because of the limited resources, particularly with regards to computational power, memory, and energy, that embedded systems often provide. Indeed, such FSM may contain several hundred states and several thousand complex transitions, making the size of corresponding parsers too large (several dozen kilobytes) for embedded systems. To simplify parser construction, automatic approaches including Gapa [1] and Zebu [2] have been proposed for generating a FSM implementation from a high-level specification of a protocol. However, to the best of our knowledge, existing automatic approaches do not address embedded systems requirements and, in particular, have not explored the use of a dedicated hardware to improve their performances, i.e., the resulting generated code is still CPU intensive.

Implementing a FSM using a dedicated hardware architecture improves performance compared to a software-based implementation. Indeed, a hardware parser can be designed specifically to execute multiple computations in parallel, in one processor clock cycle. Moreover, conditional jumps, which are massively used in software implementation of FSM, are processed in one clock cycle without pipeline break penalties. Finally, a hardware-based FSM requires a lower working frequency to reach the same performance than its software counterpart, and thus consumes less energy.

Nonetheless, developing a network application that uses hardware parsers is challenging, requiring not only expertise in hardware design and integration, but also a substantial knowledge of the protocols involved and an understanding of low-level network programming. These issues are challenging to take into account individually, and the need to address all of them at once makes hardware protocol message parsers development particularly difficult.

In this paper, we propose a co-design based architecture and a generative approach for building and using hardware parsers in a network application. To this end, we present a domain-specific language, Zebra, for describing standard text-based protocol message formats and related processing constraints. Zebra is an extension of ABNF [6], the variant of BNF used in RFCs to specify the syntax of network protocol messages, implying that the programmer can simply copy a network protocol message grammar from an RFC to begin developing a parser. It extends ABNF with annotations indicating which message fragments should be stored in data structures, and other semantic information.

A Zebra specification is processed by a compiler that generates both the HDL source code of the hardware parser implemented as a FSM, and the associated C code to drive it. The application runs on top of a middleware that hides low-level details to developers and manages the generated hardware parsers. The contributions of this paper are as follows:

- We have designed and implemented a generative approach for building hardware parsers for embedded systems. Our approach is based on a co-design architecture to provide hardware parsing capabilities to software applications.
- We have conducted a set of experiments on protocols such as HTTP, RTSP, SIP, and SMTP to assess our approach.

Preliminary results show a speedup of message parsing from 3.9 to 11 compared to software-based parsers.

The remainder of this paper is structured as follows. Section II presents the Zebra hardware platform designed to support the execution of message parsers, the middleware to manage the underlying hardware units, and the Zebra language to describe message formats, and its compiler that produces necessary HDL and C code. Section III presents the performance evaluation of Zebra-based parsers. Finally, Section IV reviews related research works and Section V concludes the paper and discusses future work.

II. Zebra Approach

The most efficient way to implement an embedded system application is to develop a fully-customized architecture, using programmable logic devices or even dedicated *Application-Specific Integrated Circuits* (ASIC). However, hardware design is a tedious and time consuming process compared to traditional software development. To alleviate the burden in hardware-based implementations, the co-design methodology proposes to slice an application based on performances it requires. Parts of the application that require high performance are implemented using dedicated hardware units. Less sensitive performance parts are implemented as software code running over a general-purpose micro-processor. Typically, the lowest part of a network application, known as the protocol-handling layer, consumes 25% of the total message processing time [5], [19]. This layer must thus be efficient and calls for a hardware-based implementation to reach the expected level of performance. To do so, we have developed the Zebra approach dedicated to building of efficient network message parsers. Our approach consists of a hardware platform to support parser execution and a middleware to transparently integrate hardware parsers into network applications. The main objective of Zebra is to minimize the need for developer intervention in the complex process of developing and using a hardware parser in a software application. Accordingly, Zebra provides a high-level specification language to describe text-based protocol message formats and related processing constraints. From this specification, a compiler automatically generates both the HDL synthesisable specification to be plugged in the hardware platform and the associated C code tailored to application needs. Figure 1 illustrates our approach.

We now describe in more details the Zebra approach. First, we describe the hardware platform we have designed to support the execution of message parsers. We then present the middleware we have developed to drive the underlying hardware-dedicated units and interconnect them with the network application running on top of a general-purpose micro-processor. Finally, we introduce the Zebra language to describe message formats, and its compiler that produces necessary HDL and C code.

A. Zebra Hardware Platform

We have combined the micro-processor and parsing units into one chip (SoC) to: (i) reduce power consumption, (ii) simplify board layout, (iii) preserve signal integrity, (iv) avoid electromagnetic interference and, (v) allow very fast communication links between them. We use *Field Programmable Gate Array* (FPGA) devices for system integration since they are particularly suitable for embedded system prototyping [4]. However, proposed approach is not limited to FPGA devices and can be easily extended to ASIC targets.

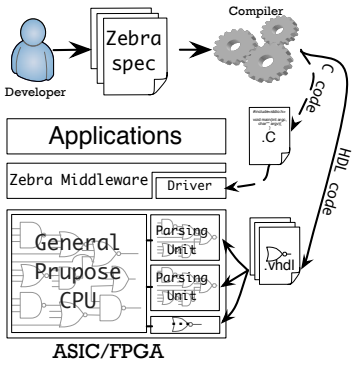

Fig. 1. Zebra approach

As illustrated in Figure 1, the Zebra platform consists of a general-purpose micro-processor to execute the application logic, and a set of dedicated hardware units for message parsing. Our current implementation relies on a LEON3 soft CPU core, which is an open-source implementation of the SPARCv8 32-bit architecture, allowing its instruction set to be extended. The use of such a soft CPU core, combined with the generation of generic HDL code, enables to implement our system on any ASIC or FPGA target, without any change.

In Zebra, parsing units are implemented as co-processors, interconnected with the micro-processor through a set of dedicated links. Particularly, a parsing unit has a specific design that includes: a 32-bit data input interface for receiving data stream to parse from the micro-processor, a 32-bit data output interface to send back parsing results, a set of both dedicated interfaces and control signals for managing the parser. The 32-bit data interfaces enable up to 4 bytes transfer per micro-processor clock cycle. The instruction set of the micro-processor has been extended to provide commands and read/write operations to each parsing unit. The number of parsing units that can be embedded depends on the size of the FPGA device and the complexity of the protocol state machines.

B. Zebra Middleware

To process network messages, an application registers a call-back function to the Zebra middleware, gives the input stream from which reading data, the protocol to use, and additionally some optional parameters. The Zebra middleware manages registered applications by reading data on input streams as they are received and sending these data to the corresponding parsing unit. The middleware then reads parsing results from the output interface of the parsing unit. When the parsing of a message element is completed, the middleware executes ad-hoc code to make the value accessible by the application. Note that the middleware can perform other computations

while waiting for the parsing units to complete their work. To increase sharing of parsing units between several tasks, the middleware seamlessly save and restore parser state when required. This context switch on the hardware parsing units is very efficient and requires only about 9 micro-processor cycles.

The Zebra middleware has been implemented in C and successfully cross-compiled for the LEON3 micro-processor. Additionally, we have modified the *gcc* toolchain to support the extended instruction set that we have introduced for controlling parsing units.

C. Zebra Language

The Zebra language is based on the ABNF notation used in RFCs to specify the syntax of protocol messages to ease its adoption by network application developers. Once having created a basic Zebra specification, the developer can further annotate it according to application-specific requirements. Figure 2 shows an excerpt of the Zebra specification for the HTTP protocol as defined in RFC 2616.

```
Request         = Request_Line
                  (( general_header
                   | request_header
                   | entity_header ) CRLF)*
                  CRLF
                  message_body? {clen} ❸;
Request_Line    = Method SP Request_URI ^uri ❷ SP
                  HTTP_Version CRLF;
Request_URI     = '*' | absoluteURI | abs_path | authority;
entity_header   = Allow
                | Content_Length
                | ...
Content_Length = 'Content-Length: ' digit+ ^clen as uint32 ❶;
```

Fig. 2. Excerpt of Zebra specification for HTTP

Annotations define the message view available to the application, by indicating the message elements that this view should include. These annotations drive the generation of the data structure that contains the message elements. For example, three message elements are annotated in Figure 2. To make an element available, the programmer only has to annotate it with the ^ symbol and the name of a field in the generated data structure that should store the element's value. For instance, in Figure 2, the Zebra programmer indicates that the application requires the URI of the request line (❷). Hence, the data structure representing the message will contain one string field: `uri`.

Besides tagging message elements that will be available to the application, annotations impose type constraints on these elements. This can be specified using the notation `as` followed by the name of the desired type. For example, in Figure 2, the `Content-Length` field value (❶) is specified to represent an unsigned integer of 32 bits (`uint32`). A type constraint enables representing an element as a type other than string. The use of both kinds of annotations allows the generated data structure to be tailored to the requirements of the application logic. This simplifies the application logic's access to the message elements.

In our experience in exploring RFCs, the ABNF specification does not completely define the message structure. Indeed, further constraints are explained in the accompanying text. For example, the RFC of HTTP indicates that the length of the body of a HTTP message depends on the `Content-Length` field value. To express this constraint, the developer only has to annotate the variable-length field `message-body` (❸) with the name of the field, between curly brackets, that defines its size (*i.e.*, `clen`). Note that such fields must have be typed as an integer.

Finally, the Zebra compiler generates a hardware parser tailored to the application needs according to the provided annotations, and associated C code to drive it. The hardware parser corresponds to a FSM whose some transitions signal the start or the end of message elements annotated in the Zebra specification. Thus, when such transitions are fired, the hardware parser writes into its output interface the name of the message element being parsed, the current position of the consumed data, and if it is the start or the end. This information is then used by the Zebra middleware to execute the corresponding generated C code, enabling to extract and save the value of the parsed message element.

III. Evaluation

We have conducted a set of experiments to assess our approach. For our experiments, we use a Xilinx Virtex-5 board and a LEON3 micro-processor configured at 50MHz. We have written Zebra specifications for four of the most ubiquitous protocols on the Internet: HTTP, SMTP, SIP, and RTSP. For each of them, we have used the Zebra compiler to automatically produce the corresponding VHDL and C code. The generated VHDL code is then synthesized in the FPGA device using the Xilinx ISE toolchain. The generated C code is cross-compiled using a modified vesion of the SPARC *gcc* toolchain and plugged into the Zebra middleware.

In order to evaluate the processing time to parse an input message from either HTTP, SMTP, SIP or RTSP, we have developed a logging application, one for each protocol, that logs messages received from the network. For each application, we have implemented two versions of its parsers: one fully implemented in software and one based on Zebra. For the software-based version, we used the Ragel [17] tool to produce an optimized FSM implementation in C.

We now present a micro-benchmark for these applications using real messages. The datasets were collected in a graduate students work area in our research laboratory during 2 hours. In our experiments, a client application replays a real trace, extracting and sending each message of this trace. We have instrumented the code of the logging applications to measure the parsing time for each received message.

Figure 3 presents the results of our evaluation. We observe that the Zebra-based parsers are between 3.9 and 11 times faster than their fully software-based counterparts.

IV. Related Work

Over the last decade, many approaches have emerged to avoid the painful task of hand writing network protocol message parsers [1], [2], [10], [15]. These approaches mainly propose a three-step process: (i) describing network protocol

		Soft-based parser			Hard-based parser		
		Min	Max	Med	Min	Max	Med
HTTP	Size	330	779	557	330	779	557
	Time	10435	20383	14917	939	1771	1326
	Avg(Clk/Char)		27.3			2.4	
	Factor				**9.9**	**12.1**	**9.3**
RTSP	Size	56	210	151	56	210	151
	Time	1946	5993	4435	409	708	573
	Avg(Clk/Char)		30.6			4.4	
	Factor				**4.8**	**9.9**	**7.2**
SIP	Size	274	1357	582	274	1357	582
	Time	7879	19416	14842	731	1828	1354
	Avg(Clk/Char)		24.4			2.2	
	Factor				**9.7**	**11.7**	**11**
SMTP	Size	6	1003	27	6	1003	27
	Time	141	8422	817	116	2090	166
	Avg(Clk/Char)		25.5			9.1	
	Factor				**1**	**5.5**	**3.9**

Size in number of characters ; Soft and Hard measures in CPU cycles.

Fig. 3. Zebra-based parsers and soft-based parsers comparison

messages in a high-level specification, (ii) generating software parsers from this high-level specification, and, (iii) providing a framework to ease the development of applications on top of generated parsers. However, none of these approaches specifically targets highly constrained embedded systems. For instance, sensor networks relying on dedicated hardware such as ASIC or FPGA do not have enough energy, code, and memory to support the aforementioned approaches.

To overcome this issue, one emerging solution is to implement parsers directly in hardware. Hence, high-level specifications of network protocol messages are mapped directly into hardware description languages such as VHDL to be then successively synthesized into ASIC or FPGA [12], [13], [14]. However, hardware parsers are provided *as is* and require strong understanding of hardware design fundamentals to integrate them with network programming applications. In contrast, the Zebra approach covers the development life-cycle of a network message parser, from its specification to the generation of hardware accelerators, to its integration into network applications.

Many commercial and academic High-Level Synthesis (HLS) tools have been proposed to generate hardware architectures from algorithmic descriptions written in C, C++, or SystemC [3], [7], [11]. However, these tools remain general purpose and are mostly oriented to datapath applications [11]. Thus, they do not provide good results for control applications, such as protocol message parsers. For example, hardware parsers generated using the LegUp tool [3] from software-based parsers used in our evaluation are at least 4.5 times slower than their Zebra-based counterparts, and consume up to 50 times more hardware resources.

To the best of our knowledge, Zebra is the only one solution that bridges the gap between HDL designs and system software engineering in the context of control applications for embedded systems.

V. Conclusion and Future Work

In this paper, we have presented Zebra, a generative approach for building hardware parsers for embedded systems. We have conducted a set of experiments on four commonly used protocols to assess our approach. Preliminary results, using micro-benchmarks, show that Zebra-based parsers are up to 11 times faster than software-based only parsers.

We are currently investigating the dynamic reconfiguration capabilities of FPGA to update at run-time the protocols supported by Zebra. We are also extending the Zebra middleware to provide advanced scheduling of available parsing units based on active clients to reduce cache misses when accessing received buffered messages stored in central memory.

References

[1] N. Borisov, D. J. Brumley, H. J. Wang, J. Dunagan, P. Joshi, and C. Guo. A Generic Application-Level Protocol Analyzer and its Language. In *14th Annual Network & Distributed System Security Symposium*, 2007.

[2] L. Burgy, L. Réveillère, J. Lawall, and G. Muller. Zebu: A Language-Based Approach for Network Protocol Message Processing. *IEEE Transactions on Software Engineering*, 37:575–591, 2011.

[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 33–36, New York, NY, USA, 2011. ACM.

[4] R. Cofer and B. F. Harding. *Rapid System Prototyping with FPGAs: Accelerating the design process*. Newnes, 1st edition, 2005.

[5] M. Cortes and J. R. Ensor. Narnia: A virtual machine for multimedia communication services. In *Proceedings of the Fourth International Symposium on Multimedia Software Engineering*, pages 246–254, 2002.

[6] D. Crocker, Ed. and P. Overell. RFC 2234: Augmented BNF for syntax specifications: ABNF, 1997. Status: PROPOSED STANDARD.

[7] M. Fingeroff. *High-Level Synthesis Blue Book*. Xlibris Corporation, 2010.

[8] S. Krishnamurthy. TinySIP: Providing seamless access to sensor-based services. In *3rd International Conference on Mobile and Ubiquitous Systems: Networking and Services*, number 4611 in Lecture Notes in Computer Science, pages 1–9, 2006.

[9] S. Leggio, J. Manner, A. Hulkkonen, and K. Raatikainen. Session initiation protocol deployment in ad-hoc networks: A decentralized approach. In *2nd International Workshop on Wireless Ad-hoc Networks*, 2005.

[10] A. Madhavapeddy. *Creating High-Performance, Statically Type-Safe Network Applications*. PhD thesis, Cambridge University, 2007.

[11] G. Martin and G. Smith. High-Level Synthesis: Past, Present, and Future. *IEEE Design & Test of Computers*, 26(4):18–25, July 2009.

[12] A. Mitra, M. R. Vieira, P. Bakalov, W. A. Najjar, and V. J. Tsotras. Boosting XML Filtering with a Scalable FPGA-based Architecture. *CoRR*, abs/0909.1781, 2009.

[13] J. Moscola, J. W. Lockwood, and Y. H. Cho. Reconfigurable Content-based Router using Hardware-Accelerated Language Parser. *ACM Transactions on Design Automation Electronic Systems*, 13(2):28:1–28:25, 2008.

[14] J. Öberg, A. Hemani, and A. Kumar. Grammar-Based Hardware Synthesis from Port-Size Independent Specifications. *IEEE Transactions on Very Large Scale Integration Systems*, 8(2):184–194, 2000.

[15] T. Stefanec and I. Skuliber. Grammar-based SIP Parser Implementation with Performance Optimizations. In *Proceedings of the 11th International Conference on Telecommunications, ConTEL '11*, pages 81–86, 2011.

[16] P. Stuedi, M. Bihr, A. Remund, and G. Alonso. SIPHoc: Efficient SIP middleware for ad hoc networks. In *Proceedings of the 8th ACM/IFIP/USENIX International Conference on Middleware*, 2007.

[17] A. D. Thurston. Parsing Computer Languages with an Automaton Compiled from a Single Regular Expression. In *Proceedings of the 11th International Conference on Implementation and Application of Automata*, CIAA'06, pages 285–286, Berlin, Heidelberg, 2006. Springer-Verlag.

[18] B. Upender and P. Koopman. Communications protocols for embedded systems. *ACM Transactions on Programming Languages and Systems*, 11(7):46–58, 1994.

[19] S. Wanke, M. Scharf, S. Kiesel, and S. Wahl. Measurement of the SIP parsing performance in the SIP Express Router. In *Dependable and Adaptable Networks and Services*, number 4606 in Lecture Notes in Computer Science, pages 103–110, 2007.