Improving the Performance of Message Parsers for Embedded Systems

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

Improving the Performance of Message Parsers for Embedded Systems

Jigar Solanki
LaBRI, University of Bordeaux
Talence, France
jigar.solanki@labri.fr

Laurent Réveillère
LaBRI, University of Bordeaux
Talence, France
laurent.reveillere@labri.fr

Yérom-David Bromberg
LaBRI, University of Bordeaux
Talence, France
david.bromberg@labri.fr

Bertrand Le Gal
IMS, University of Bordeaux
Talence, France
bertrand.legal@ims-bordeaux.fr

Tégawendé F. Bissyandé
LaBRI, University of Bordeaux
Talence, France
tegawende.bissyande@labri.fr

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 systematically outperform software-based parsers.

1. 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 [23]. 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 [12] and mobile ad-hoc networks [13, 21].

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 [3] and Zebu [4] 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. For instance, hardware design relies on low-level languages (HDL) such as VHDL [2] or Verilog [1] whereas application software is often based on widely used programming languages like Java or C. 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 [10], 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 up to 11 compared to equivalent parsers fully implemented in software, and up to 1850 compared to parsers extracted from widspread deployed Internet servers.

The remainder of this paper is structured as follows. Section 2 describes the callenges of building network applications for embedded systems. Section 3 introduces 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 4 assesses the performance and memory footprint of Zebra-based parsers compared to parsers fully implemented in software. Finally, Section 5 presents related work and Section 6 concludes.

2. BUILDING NETWORK APPLICATIONS FOR EMBEDDED SYSTEMS

The lowest part of a network application, known as the protocol-handling layer, enables communication between applications through application-centric protocols. It must take into account both the constraints of the protocol, which determine the message structure, and the requirements of the application, which determine how the message elements will be used. Typically, the protocol-handling layer consumes 25% of the total message processing time [8, 9, 24]. Further, the inherent complexity of network protocol message formats makes the development of this layer very difficult. When the protocol-handling layer makes available a message element to the application logic, it provides the information about the message according to a logical view. Such a presentation includes some interpretation of the message elements, for instance with each kind of message element being represented by its proper application-level type. The information about a message required by the application logic may differ according to the nature of the application being implemented. For instance, an application to retrieve the temperature from a sensor needs to look at a few message elements to perform its task. On the other hand, a logging application needs additional information such as the actual destination, the requester and some time-stamps.

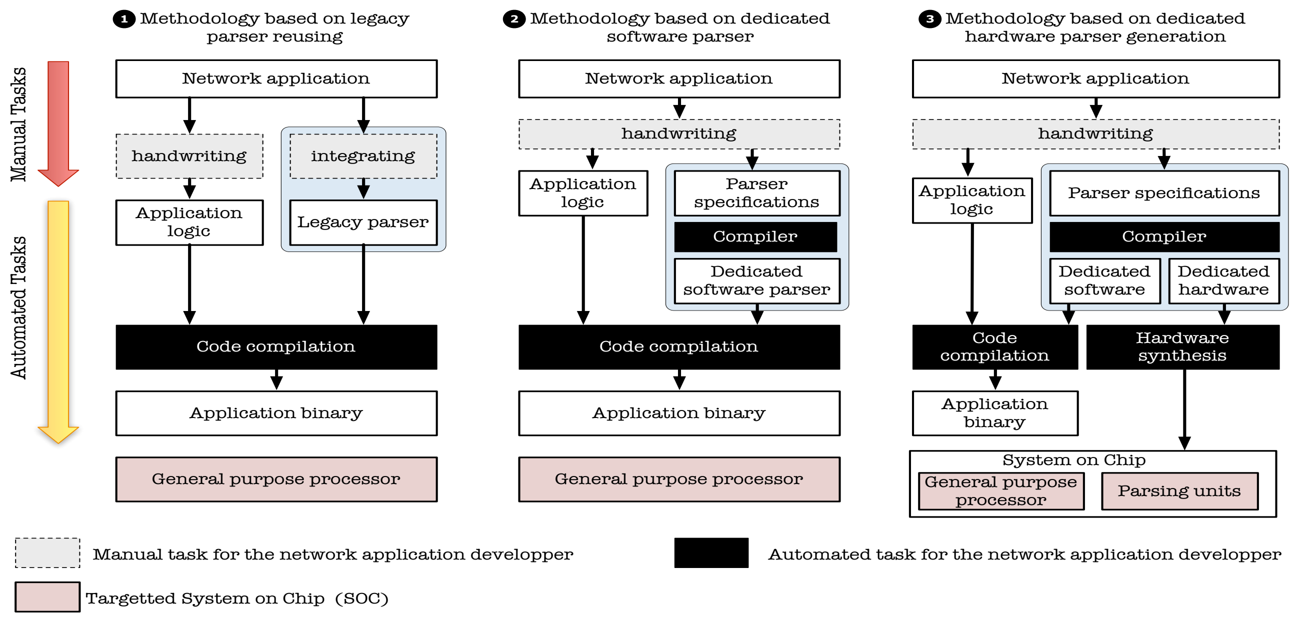
Considering the limited ressources that embedded systems often provide, developing such a network application is a challenging task. To reduce developer expertise while providing application efficiency, developers are mainly faced to deal with three development methodologies (See Figure 1). In the first methodology (Figure 1❶), developers do not have to develop from the ground up the protocol-handling layer. In fact, developers leverage on existing parsers (called thereafter legacy parsers) to process input messages. Thus, developing a network application consists in writing its application logic with some glue code to interact with the required underlying legacy parsers. If this approach drastically reduces developers' tasks, it lacks in terms of efficiency. Legacy parsers are general-purpose: they are as much as possible compliant to standard protocols without being optimized for specific needs of a particular application. To overcome this drawback, in the second methodology (Figure 1❷), a protocol-handling layer relies on a dedicated software parser specifically designed according to the application being developed. Such parsers are generated *via* a compiler that takes as input a high level specification. Such specifications indicate which message fragments, and other semantic information are required for the application logic. Thus, the complexity of the underlying FSM of generated parsers is reduced, and thus their overall efficiency is improved as such parsers parse only the required message fragments as opposed to legacy parsers that parse all message fragments whatever the application logic.

To further improve efficiency, we argue that generated parsers can be implemented as accelerated logical units using programmable logic devices (FPGA) or even dedicated *Application-Specific Integrated Circuits* (ASIC) directly interconnected with a micro-processor through a dedicated interface. To minimize the need for developer intervention in the complex process of developing and using a dedicated hardware parser in a software application, we have enhanced the previous methodology as illustrated in Figure 1❸. From a high level specification, used to describe text-based protocol message formats and related processing constraints, a compiler generates both HDL synthesisable specification to be plugged in a hardware platform, and the associated C code tailored to application needs. Correspondingly, with this third methodology, we guaranty to developers a good trade-off between efficiency and simplicity for developing network applications.

We now describe in more details the Zebra approach [16] that follows the aforementioned third methodology. First, we introduce the Zebra language to describe message formats, and its compiler that produces necessary HDL and C code. We then describe the hardware platform we have designed to support the execution of message parsers. Finally, we 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.

3. ZEBRA APPROACH

The most efficient way to implement an embedded system application is to develop a fully-customized architecture, using FPGA or dedicated ASIC. However, hardware design



1 Methodology based on legacy parser reusing
2 Methodology based on dedicated software parser
3 Methodology based on dedicated hardware parser generation
Manual Tasks
Automated Tasks
Network application
handwriting
integrating
Application logic
Legacy parser
Parser specifications
Compiler
Dedicated software parser
Dedicated software
Dedicated hardware
Code compilation
Hardware synthesis
Application binary
General purpose processor
System on Chip
Parsing units
Manual task for the network application developper
Automated task for the network application developper
Targetted System on Chip (SOC)

Figure 1: Different methodologies to design network applications

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.

Consequently, in the following subsections, we present 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.

3.1 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 [7]. However, proposed approach is not limited to FPGA devices and can be easily extended to ASIC targets.

The Zebra platform consists of a general-purpose microprocessor 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.

3.2 Zebra Middleware

To process network messages, an application registers a callback 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.

3.3 Zebra Language

To address the difficulty of implementing network protocol message parsers in hardware, we have developed the domain-specific language Zebra. The main objective of the Zebra language is to allow the developer to specify the message syntax using a high-level notation, while minimizing the need for developer intervention in the complex process of translating this notation to synthesizable HDL code. The Zebra language is inspired by Zebu [5, 4], a language for describing HTTP-like text-based protocol message formats and related processing constraints. Accordingly, the syntax of Zebra is based on the ABNF [10] notation used in RFCs to specify the syntax of protocol messages.

3.3.1 ABNF

An ABNF specification defines the syntax of protocol messages. It consists of a set of derivation rules, each defining a set of alternatives (separated by `|`) that represent a sequence of terminals and nonterminals. ABNF also includes a general form of repetition to indicate the number of occurrences of a terminal or non-terminal. As an example, Figure 2 shows an excerpt of the ABNF specification of the HTTP protocol as described in RFC 2616. The first rule (line 3) defines a request as a request line, followed by a sequence of message headers, followed by a blank line, and then a payload (line 8). Line 9 defines the structure of a request line, comprising not only tokens, as derived by *e.g.* the nonterminal `Method`, but also withespace, as derived by `SP` and `CLRF`. The nonterminal `entity-header` (line 11) is defined as an alternation, as indicated by `|`, of the HTTP headers. Line 14 defines the Content-Length header, which is found in most HTTP messages. This header consists of the key "`Content-Length`", which is case insensitive, immediately followed by a colon and one or more digits.

```
 1  The Content-Length entity-header field indicates the size of
    the entity-body, [...]
 2
 3  Request = Request-Line                        ; Section 5.1
            *(( general-header                    ; Section 4.5
 5             | request-header                   ; Section 5.3
 6             | entity-header ) CRLF)            ; Section 7.1
 7          CRLF
 8          [ message-body ]                      ; Section 4.3
 9  Request-Line   = Method SP Request-URI SP HTTP-Version CRLF
10  Request-URI    = "*" | absoluteURI | abs_path | authority
11  entity-header  = Allow                        ; Section 14.7
12                 | Content-Length               ; Section 14.13
13                 | ...
14  Content-Length = "Content-Length" ":" 1*DIGIT
```

Figure 2: Excerpt of ABNF specification from RFC 2616

3.3.2 Annotating an ABNF specification

To create a zebra specification, the developer only needs to slightly refactor the ABNF specification. Reserved characters have to be protected and repetitions need to be expressed using postfix operators: `?` (optional), `*` (any), and `+` (some). Figure 3 illustrates the zebra specification for the HTTP protocol. Once having created a basic Zebra specification, the developer can further annotate it according to application-specific requirements.

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 of the parser. For example, three message elements are annotated in Figure 3. To make an element available, the programmer only has to annotate it with the ˆ symbol and the name of a field in the generated data structure that should store the element's value. For instance, in Figure 3, 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`.

```
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;
```

Figure 3: Excerpt of Zebra specification for HTTP

Besides tagging message elements that will be available to the application, annotations impose type constraints on these elements (❶). This can be specified using the notation `as` followed by the name of the desired type. For example, in Figure 3 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 (Figure 2, line 1). 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 define its size (*e.g.*, `{clen}` (❶)). Note that such fields must have be typed as an integer.

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. Note that a transition in the FSM is evaluated in only one single clock cycle.

4. EXPERIMENTS

.

We have conducted a set of experiments to assess our approach. For our experiments, we use a ML507 development board from Xilinx^{TM} that includes a Virtex-5 FPGA device. To allow general-purpose software code to be executed, we have integrated a Leon-3 micro-processor configured at 50MHz (*sparc* processor) in the FPGA device. The processor runs a Linux 2.6.36 kernel. System-on-Chip implementation on the FPGA circuit are realized using the Xilinx ISE toochain.

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 three versions: one fully implemented in software-based on legacy parsers, one fully implemented in software-based on parsers dedicated to application needs, and one based on Zebra. Legacy parsers have been exctracted from widspread deployed Internet servers: Cherokee[1] for HTTP, LibOSIP[2] for SIP, GSTreamer[3] for RTSP, and Postfix[4] for SMTP. For the software-based versions based on parsers dedicated to application needs, we used the Ragel [22] tool to produce an optimized implementation.

We use as datasets real messages 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.

4.1 Execution time

We have instrumented the code of the logging applications to measure the parsing time for each received message. We consider the median time over several successive executions. Figure 4 presents the results of our evaluation with various message sizes. We observe that legacy parsers do not compete compared to application specific parsers. They are at least 9 times slower than dedicated parser generated by Ragel, as in the case of HTTP, and up to 185 times slower in the case of SIP. Indeed, legacy parsers are designed to fully support the various features of a protocol. Their general-purpose nature make them usable in many different application domains, including our logging applications, but at the expense of a significant performance overhead.

Zebra-based are even more efficient than dedicated parsers fully implemented in software. They are at least 2 times faster, as in the case of SMTP, and up to 11 times faster in the case of HTTP. When compared to legacy parsers, Zebra-based parsers are from 68 times faster, as in the case of SMTP, to 1850 times faster in the case of SIP.

4.2 Static memory footprint

We now consider the static memory requirements of logging applications when compiled for the Leon-3 processor.

[1]Cherokee: http://www.cherokee-project.com/
[2]LibOSIP: http://www.gnu.org/software/osip/
[3]GSTreamer: http://www.gstreamer.net/
[4]Postfix: http://www.postfix.org/

		Legacy parser			Dedicated parser			Zebra-based parser		
		Min	Max	Med	Min	Max	Med	Min	Max	Med
HTTP	Size	428	437	557	428	437	557	428	437	557
	Time	183476	227678	231320	14050	14053	23274	1291	1317	1975
	Avg		451.8			36.1			3.2	
	Factor	**92**	**175**	**139**	**10**	**12**	**11**	**1**	**1**	**1**
RTSP	Size	56	210	151	56	210	151	56	210	151
	Time	168802	132726	145920	3098	5456	6941	537	741	848
	Avg		1072.9			37.1			5.1	
	Factor	**172**	**314**	**179**	**5**	**8**	**7**	**1**	**1**	**1**
SIP	Size	274	1244	581	274	1244	581	274	1244	581
	Time	1286710	2955595	2779204	9766	18637	16996	1001	1824	1548
	Avg		3345.1			25.6			2.4	
	Factor	**1285**	**2043**	**1792**	**9**	**11**	**10**	**1**	**1**	**1**
SMTP	Size	8	420	202	8	420	202	8	420	202
	Time	17162	650767	346034	2093	5482	3683	1134	1920	1522
	Avg		1609.5			17.9			9.8	
	Factor	**15**	**413**	**247**	**1**	**2**	**2**	**1**	**1**	**1**

Size in number of characters ; measures in CPU cycles.

Figure 4: Legacy, dedicated and Zebra-based parsers comparison

This includes the memory requirements of Linux, the size of the compiled message parser, and the size of the compiled application logic. Without any application installed and when running a minimal root filesystem, Linux uses 1.8MB. The size of the compiled message parsers across the various versions is shown in Table 5. The compiled Zebra-based generated code is only 1.8KB because most of the complexity of message parsing is shift to hardware parsing units. The last two columns also show the size occupation of these parsing units and the size of the Leon-3 processor on the FPGA.

	Legacy	Dedicated	Zebra-based	FPGA occupation	
	parser	parser	parser	parser	processor
HTTP	188.2	13.0	1.8	3%	30%
SMTP	457.7	81.4	1.8	7%	30%
SIP	183.5	31.1	1.8	12%	30%
RTSP	241.8	12.6	1.8	4%	30%

Figure 5: Compiled code size of parsers, in kilobytes

5. RELATED WORK

Over the last decade, many approaches have emerged to avoid the painful task of hand writing network protocol message parsers [3, 4, 14, 20]. These approaches mainly propose a three-step process: (i) describing network protocol 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 [17, 18, 19]. 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 lifecycle 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 [6, 11, 15]. However, these tools remain general purpose and are mostly oriented to datapath applications [15]. Thus, they do not provide good results for control applications, such as protocol message parsers. For example, hardware parsers generated using the LegUp tool [6] 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.

6. 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. We show that Zebra-based parsers are at least 68 times faster than legacy parsers and at least 2 times faster than dedicated parsers generated by Ragel. In addition to improving execution time of message parsing, Zebra-based parsers requires only few bytes of static memory, most of the processing being shift to hardware accelerated parsing units.

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.

7. REFERENCES

[1] Ieee standard hardware description language based on the verilog(r) hardware description language. *IEEE Std 1364-1995*, 1996.

[2] Ieee standard vhdl language reference manual. *IEEE Std 1076-2000*, 2000.

[3] 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.

[4] 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.

[5] L. Burgy, L. Réveillère, J. L. Lawall, and G. Muller. A language-based approach for improving the robustness of network application protocol implementations. In *26th IEEE International Symposium on Reliable Distributed Systems*, pages 149–158, Beijing, Oct. 2007.

[6] 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.

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

[8] M. Cortes, J. Ensor, and J. Esteban. On SIP performance. Technical report, Bell Labs Technical Journal 3, 2004.

[9] 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.

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

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

[12] 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.

[13] 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.

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

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

[16] J. Mercadal, L. Reveillere, Y. Bromberg, B. Le Gal, J. Solanki, and T. Bissyande. Zebra: Building efficient network message parsers for embedded systems. *Embedded Systems Letters, IEEE*, 4(99):69–72, 2012.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.