System Architecture for XML Offload to a Cell Processor-Based Workstation

Stefan **Letz**

Roland **Seiffert**

Jan **van Lunteren**

Paul **Herrmann**

Abstract

This paper describes the design, prototype implementation, and evaluation of a system architecture for XML offload to a Cell processor-based workstation. This architecture includes a high-performance parser based on a novel enhanced finite state machine technology.

Table of Contents

1. Introduction

Cell is a new processor architecture jointly developed by Sony, Toshiba, and IBM. The architecture's first processor combines one Power Architecture^{TM} and eight Single Instruction Multiple Data cores, the so-called Synergistic Processing Elements, in a “supercomputer on a chip”. Besides its intended use in next-generation game consoles, high-definition television sets, and Cell processor-based workstations, the processor is also interesting for other application scenarios. One promising idea is to exploit Cell for function offload, where computationally-intensive processing tasks are moved from application systems to specialized offload systems, enabling existing applications to benefit from the computing power of Cell.

An especially pressing subject in today's application environments is the acceleration of XML processing. XML has emerged as the de facto standard for exchanging and handling data and information, but its processing is still plagued by a tremendous lack of performance. In particular, parsing, the very basic step of XML processing, considerably increases the load on application systems that are already operating at their limits [1].

This paper describes the design, prototype implementation, and evaluation of a system architecture for XML offload to a Cell processor-based workstation, bringing together a novel powerful processor technology and an increasingly important software technology that suffers from its performance problem [2].

2. Cell Overview

The Cell processor is the result of a joint development project of Sony, Toshiba, and IBM. In 2001, the three formed the so-called STI Design Center in Austin, Texas, whose task was to create a new powerful processor architecture.

2.1. Cell Processor Architecture

The first processor of the Cell architecture is a multi-core design, comprising nine cores [3]. The chip's main unit is a 64 bit Power Architecture^{TM} core, called Power Processor Element (PPE). The PPE has an in-order pipeline consisting of eleven stages, provides Simultaneous Multi-Threading (SMT), and has a Vector Multimedia Extension (VMX) unit. The PPE's purpose is to run the operating systems and orchestrate the other units.

These other units are eight Single Instruction Multiple Data (SIMD) cores, so-called Synergistic Processor Elements (SPEs). Compared with the PPE, they are relatively simple, but allow fast fixed-point and floating-point operations on multiple data sets simultaneously. Each SPE has a 256 KB memory, the so-called Local Store (LS), which can be used freely by code running on the SPE and holds both program code and data.

All nine cores are connected via the Element Interface Bus (EIB), a high-bandwidth data ring that allows very fast communication between the units and supports over 100 outstanding Direct Memory Access (DMA) requests, which transfer data between the Local Stores and system memory. This powerful bus gives the architecture its second name: Broadband Processor Architecture. Furthermore, the external FlexIO bus allows multiple Cell chips to be connected.

With its design, the chip is especially well suited for applications such as high-performance computing, graphics and visualization, multimedia processing, and digital entertainment. But besides these obvious uses, others may be interesting as well.

2.2. Cell Processor-Based Workstation

As has been publicly announced, IBM is building a workstation based on the Cell processor. The exact form and system features have not yet been made public. The prototype of this workstation is an IBM BladeCenter$^{®}$ server.

3. XML Offload

The basic idea of XML offload is to move typically computationally-intensive XML processing tasks from systems that run the applications to other designated systems. Possible motivations for doing so are the following:

1. Reducing the workload of the application systems, i.e., freeing potentially expensive resources such as CPU cycles, memory, or disk space.
2. Accelerating processing.
3. Providing specialized services that otherwise would not be possible.
4. Providing centralized services on the offload systems, possibly using approaches such as load balancing or accounting.

Most existing offload solutions originate from motivations 1 and 2, e.g., hardware systems such as XML accelerator PCI cards. The work presented here is also driven by these two motivations, but is characterized by sufficient flexibility to consider also points 3 and 4 above.

3.1. Concepts

One can basically distinguish two concepts used in offload solutions: the client-server concept and the proxy concept.

The main idea of the client-server concept is to have a server provide services that are requested by a multitude of clients. When viewed from a client-centric perspective, or when a server serves only one client, this concept can also be interpreted as the server functioning as a co-processor for the client. A client-server infrastructure basically consists of four components:

- At least one client, i.e., an application system that offloads processing tasks.
- One or more servers, i.e., designated systems that process these tasks.
- One or more services provided by the servers.
- A communication protocol between clients and servers.

In contrast to extending a system architecture by introducing a server system and explicitly offloading processing tasks to this system, the proxy concept adds a proxy system between existing communication steps. A proxy infrastructure can be divided into the following components:

- One or more clients, i.e., application systems.
- One or more proxies, which implicitly handle processing tasks on the in- or outbound communication of the clients.
- A communication protocol between clients and proxies.
- A communication protocol between proxies and external systems.

3.2. Existing Solutions

There are numerous existing solutions for XML offload. For example, the Tarari RAX Content Processor (RAX-CP) is a hardware solution for XPath-based processing of XML data, having the form of a standard PCI card and functioning as an XML co-processor [4]. The underlying technique, called “Simultaneous XPath”, allows evaluation of given XML data using multiple XPath expressions. Tarari promises large performance improvements over software-only processing approaches. Actually, the programmable logic of Tarari's PCI card allows solutions for handling tasks other than XML processing.

Two XML acceleration appliances come from DataPower and are called XA35 XML Accelerator^{TM} and XS40 XML Security Gateway^{TM}. Whereas the XA35 handles XSLT transformations, the XS40 provides Web Services Security (WS-Security) support. Both appliances are network-attached hardware accelerators and can operate in co-processor or proxy mode. DataPower claims to deliver “wire-speed” functionality.

4. System Infrastructure

The system architecture presented in this paper consists of two elements: a basic infrastructure and a number of specialized offload services integrated therein. The system infrastructure provides means to move processing tasks from application systems to designated offload systems. Following the general considerations of Section 3.1, its design and implementation are divided into three components: a client, a server, and a communication interface. This section describes the prototype implementation of the system infrastructure.

4.1. Design Objectives

In contrast to the examples of offload solutions presented in Section 3.2, this work exploits a general-purpose processor architecture instead of special-purpose hardware or programmable logic. The focus is not on high-level XML standards, but on XML parsing as the basic step of XML processing. Furthermore, special attention is paid to providing a more general system infrastructure for offload that is not limited to parsing.

The most important design objectives are as follows:

- *Transparent integration and portability*. The client should transparently integrate into existing application programming interfaces. Furthermore, it should be implemented in a portable fashion.
- *Flexibility*. The server should be able to provide various offload services besides XML parsers. The communication interface should be independent from the services that use it.
- *Efficiency and performance*. The overhead introduced by the infrastructure should be as small as possible. Furthermore, the server’s underlying hardware should be used optimally in order to provide maximum processing performance. The communication interface should allow an efficient, low-latency implementation on both client and server. It should enable the server to mimic the behavior of local functionality on the application systems. When parsing XML, a local SAX parser returns the first SAX event as soon as it has parsed enough input data. Thus, the server should be able to send the first parsing results to the client as soon as possible.

4.2. Prototype Implementation

To achieve these design objectives, the client is implemented in Java^{TM} and integrates into the Java^{TM} API for XML Processing (JAXP). If an offload server is available, the client uses an XML parser service on the server. Otherwise, the client uses a local parser.

The server is implemented in C++ on the Linux^{TM} operating system. It supports multiple services at the same time that can provide functionality other than XML parsing. A specific service is identified by a unique hierarchical name. Each instance of a service provides a number of properties, which can be set and retrieved by the client. The prototype implementation of the server contains SAX and DOM parser services based on Expat, Xerces-C++, and the XML accelerator described below.

The so-called event-based communication interface provides asynchronous, parallel communication by implementing streaming. The client streams input data to the server, and the server streams the processing results back to the client.

The communication interface defines some common communication primitives and allows services to add specific ones.

4.3. Performance Evaluation

The performance evaluation of the infrastructure reveals an unexpected bottleneck in existing Java^{TM} parser interfaces. These interfaces require an inefficient mapping of the communication interface to their structures, so that parsing using a parser service takes about as long as parsing using a local parser, e.g., Xerces-J, does and causes approximately the same amount of CPU load. Figure 1 compares the performance of SAX parsing using the Xerces-C++ parser service and using a local Xerces-J parser, where the parser instances are reused to process each XML file 50 times.

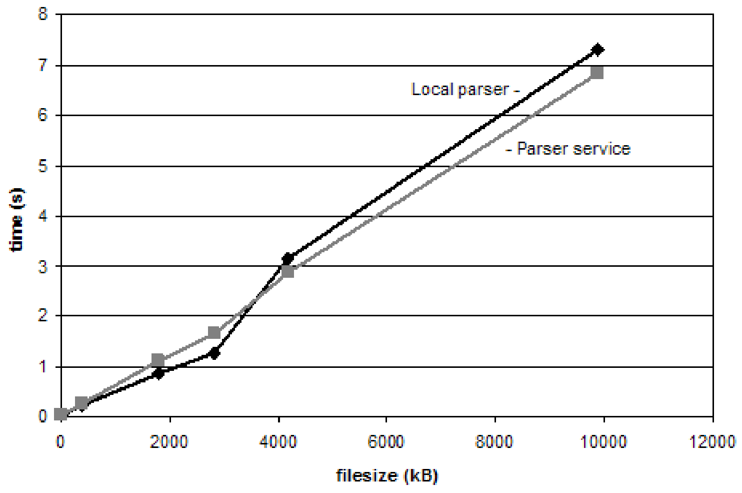

Figure 1. SAX Performance Evaluation

The main causes of this bottleneck are inherent restrictions of the Java^{TM} programming language, such as the absence of C++-style pointers and memory operations. A potential solution to this problem could be the integration of the offload infrastructure through the Java^{TM} Native Interface (JNI). Another approach could be the modification of the Java^{TM} Runtime Environment (JRE) to provide highly efficient data types for the interfaces needed. However, at the time of writing, we have not yet obtained conclusive results, so that further work is necessary.

5. XML Accelerator

This section describes the main implementation concepts of the accelerator parser service that is executed on the Cell processor, which exploits a novel technology called BaRT-based Finite State Machine (B-FSM).

5.1. Enhanced FSM Technology

A key aspect of the B-FSM technology is that it is based on state transition rules that include conditions for the current state and input values, and have been assigned priorities. In each cycle, the highest-priority transition rule is determined that matches the actual values of the state and input. This rule is then used to update the current state and to generate output. This concept will now be explained using the example of a state transition diagram, shown in Figure 2, that detects the first occurrence of one of the two patterns “121h” and “ABh” in hexadecimal notation. The state transition diagram can be described using the set of transition rules given in Table 1.

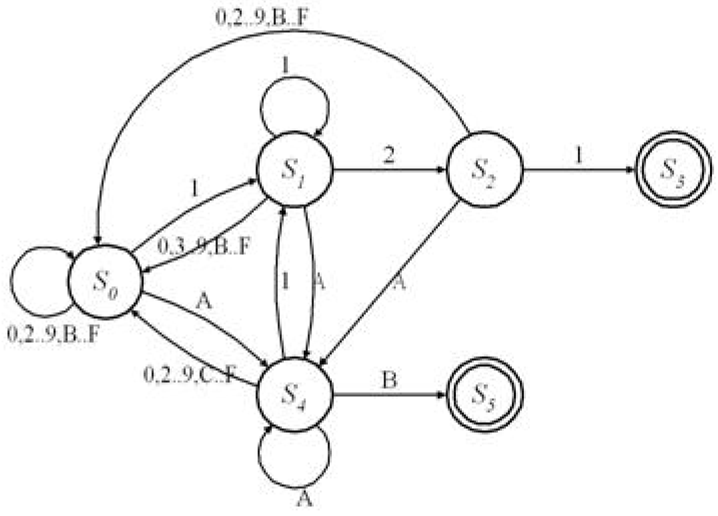

Figure 2. Sample State Transition Diagram

rule	current state	input	→	next state	priority
R_0	*	*	→	S_0	0
R_1	*	1h	→	S_1	1
R_2	S_1	2h	→	S_2	1
R_3	S_2	1h	→	S_3	2
R_4	*	Ah	→	S_4	1
R_5	S_4	Bh	→	S_5	1

Table 1. Transition Rules

Transition rule R_2 specifies that if the current state equals S_1 and the input equals '2h', a transition will be made to state S_2. Transition rules R_1 and R_3 specify that with an input symbol '1h', a transition will be made to state S_3 if the current state equals S_2 and that a transition will be made to state S_1 if the current state is any state other than S_2. This is achieved by including a wildcard condition for the current state (represented by a symbol '*') in rule R_1 and by assigning a higher priority to rule R_3 than to rule R_1. Transition rule R_0, involving wildcard conditions for both the state and the input and having a minimum priority, can be regarded as a default rule that is only selected if no other matching rule can be found. These six transition rules describe the entire state transition diagram, as can be verified in Figure 2.

The B-FSM concept applies a hash-based search algorithm, called BaRT (Balanced Routing Table search), to find the highest-priority transition rule matching the state and input values. By exploiting BaRT in combination with several optimization techniques, including state clustering and encoding, the B-FSM technology is able to provide a unique combination of high performance, low storage requirements, and fast dynamic updates, while supporting very large state transition diagrams and wide input and output vectors, resulting in a significant gain over conventional programmable state machine technologies. The strength of the B-FSM technology is clearly illustrated by the design of a programmable pattern-matching engine that is capable of processing more than 20 Gb of data per second, by matching the data against thousands of patterns in parallel, while achieving a storage efficiency of about 1,500 patterns or 25,000 characters in only 100 KB of memory [5]. For more details on the BaRT and B-FSM technologies, the reader is referred to [6].

5.2. B-FSM-based XML Acceleration

Figure 3 shows a block diagram of the XML accelerator that is based on the B-FSM technology. It consists of two main components: the B-FSM and an instruction handler.

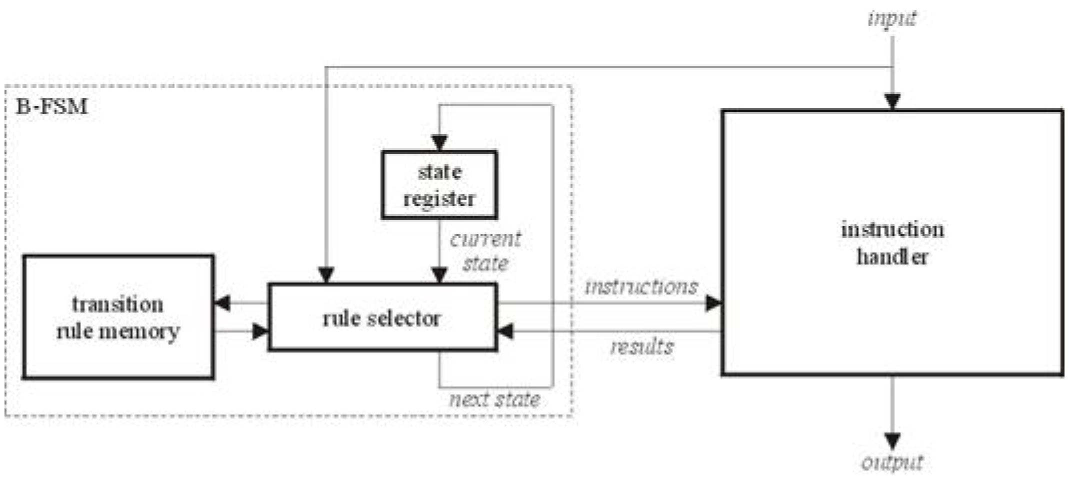

Figure 3. B-FSM-based XML Accelerator

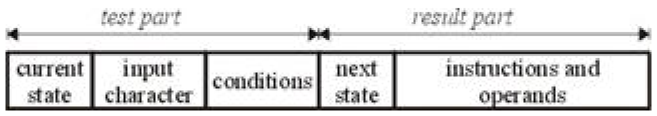

Figure 4. Transition-Rule Vector

The B-FSM operates as a programmable controller, executing programs that are specified using state transition rules contained in the transition-rule memory shown in Figure 3. Each transition rule is stored as a transition-rule vector, depicted in Figure 4, which includes a current state and input field that are tested against the actual values of the state register and input, as well as a conditions field that is used to specify additional conditions to be tested. The next-state field of the highest-priority matching transition rule that is selected by the rule selector function shown in Figure 3 is used to update the state register.

The basic concept of the accelerator is illustrated in Figure 5, which shows a simple state transition diagram containing multiple potential execution paths. The actual path taken through the diagram during program execution is determined by real-time evaluation of the conditions associated with the transition rules that define the state transition diagram. The B-FSM will dispatch the instructions and operand values that are contained in the transition rules along the path selected through the state transition diagram to the instruction handler. In this way, the B-FSM operates as a tightly-coupled controller, which is capable of evaluating multiple conditions in parallel and, in response, can immediately trigger the execution of the appropriate instructions.

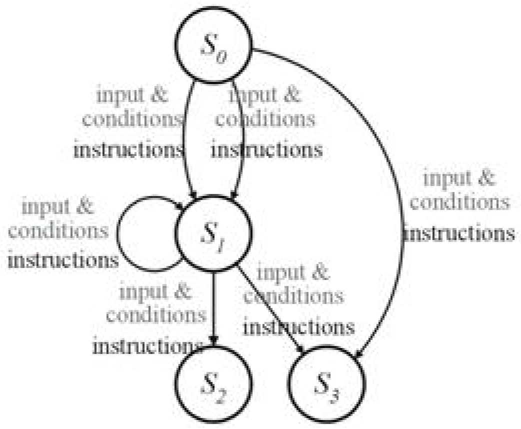

Figure 5. B-FSM-based Controller

The XML accelerator supports a variety of conditions that can be specified for each transition rule. These conditions can relate to character information, e.g., allowing direct testing of whether an input character is a legal character to be used in an XML name, whether it is a white-space character, or whether it is part of a valid hexadecimal string representation. Conditions can be specified related to the outcome of the instruction execution by the instruction handler, e.g., to determine whether a string of input characters matches a previous string of input characters that have been temporarily stored in a local memory (e.g., to check for matching start and end tags). A third category of conditions relates to the detection and handling of exception and error conditions, e.g., buffer overflows. The instruction handler supports a wide range of instructions designed for efficient XML processing, including encoding-related instructions (e.g., conversion from UTF-8 to UTF-16), storage, retrieval and comparison of character strings in a local memory, searching strings in a search structure, and a flexible generation of output data. Multiple instructions can be combined in one transition rule, so that they can be executed in parallel by the instruction handler. In addition, a set of special instructions is directly executed by the B-FSM controller and enables procedure calls to subroutines defined by a collection of transition rules. A detailed discussion of all conditions and instructions supported by the XML accelerator would exceed the scope of this paper, but will be part of a future publication.

The XML accelerator design, as described above, was initially focused on a hardware implementation that could efficiently exploit the parallelism available between the B-FSM and the various functions within the instruction handler [6]. However, it appeared that the same concept could be used to create a fast and efficient software implementation for execution on the Cell processor. This was done by means of a semi-automatic process to generate source code by efficiently combining controller and instruction-handler functions into single threads, which were then optimized for the instruction set and execution architecture of the Cell processor.

5.3. Performance Evaluation

Based on the conditions and instructions supported by the XML accelerator, an initial program has been created in the form of an enhanced state transition diagram specified by transition rules that implements the basic functionality of a non-validating SAX parser. Figure 6 shows the performance of this program, called SAXy, on different processors and in comparison to libxml based on various XML files from IBM DB2$^®$ customers.

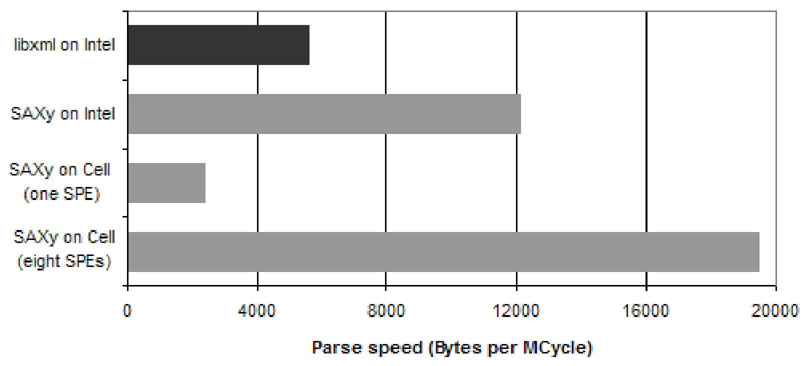

Figure 6. XML Accelerator Performance Evaluation

6. Conclusions and Outlook

The prototype implementation of the offload system infrastructure disclosed a bottleneck in existing Java^{TM} parser interfaces. This problem presents a serious obstacle for various Java^{TM}-based offload and acceleration scenarios, even beyond XML parsing. But despite its current limitations in the context discussed in this paper, the Java^{TM} environment remains especially interesting as it is widely used in XML applications.

We were, however, able to demonstrate significant performance gains for XML parsing by exploiting the Cell processor with a novel approach to parsing technology. In non-Java^{TM} scenarios, this accelerator can be used to realize high-performance XML offload and accelerator solutions. A future publication will cover the design and implementation details of the XML accelerator.

7. Trademark Notices

IBM, Power Architecture, BladeCenter, and DB2 are trademarks of International Business Machines in the United-States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Tarari is a trademark or registered trademark of Tarari, Inc. or its subsidiaries in the United States and other countries.

DataPower, XA35 XML Accelerator, and XS40 XML Security Gateway are trademarks of DataPower Technology, Inc.

Other company, product, or service names may be trademarks or service marks of others.

Acknowledgements

We thank Jochen Roth, Matthias Schütz, Michel Zedler, and Tobias Thierer for their enthusiastic work on XML acceleration using Cell technology. We thank Joseph Bostian for providing valuable help and expertise. Special thanks go to Charlotte Bolliger for her help in preparing this manuscript.

Bibliography

[1] *M. Nicola and J. John. XML Parsing: A Threat to Database Performance.* Proc. of the 12th International Conference on Information and Knowledge Management (CIKM), pages 175-178, New Orleans, LA, USA, November 2003.

[2] *S. Letz. Cell Processor-Based Workstation for XML Offload – System Architecture and Design.* University of Leipzig, Department of Computer Science, Leipzig, Germany, May 2005.

[3] *D. Pham et al. The Design and Implementation of a First-Generation CELL Processor.* Proc. of the 2005 IEEE International Solid State Circuits Conference (ISSCC), San Francisco, CA, USA, February 2005.

[4] *M. Leventhal. Random Access XML Programming Assisted with XML Hardware.* XML Conference 2004, Washington, DC, USA, November 2004.

[5] *J. van Lunteren and A.P.J. Engbersen. A High-Performance Pattern-Matching Engine for Intrusion Detection.* Hot Chips 17, Stanford University, Palo Alto, CA, USA, August 2005.

[6] *J. van Lunteren and A.P.J. Engbersen. XML Accelerator Engine.* First International Workshop on High Performance XML Processing, in conjunction with the 13th International World Wide Web Conference (WWW2004), New York, NY, USA, May 2004.

Biography

Stefan **Letz**
IBM Corporation [http://www.ibm.com]
IBM Deutschland Entwicklung GmbH [http://www.ibm.com/de/entwicklung/]
Schönaicher Strasse 220
Böblingen
71032
Germany
stefan.letz@de.ibm.com

Stefan Letz received his diploma degree in Computer Science from the University of Leipzig, Germany, in June 2005. He has been with the IBM laboratory in Böblingen, Germany, since 2003, working on database backup/restore solutions. There, he also wrote his diploma thesis entitled "Cell Processor-Based Workstation for XML Offload - System Architecture and Design".

Roland **Seiffert**
IBM Corporation [http://www.ibm.com]
IBM Deutschland Entwicklung GmbH [http://www.ibm.com/de/entwicklung/]
Schönaicher Strasse 220
Böblingen
71032
Germany
seiffert@de.ibm.com

Roland Seiffert received his diploma degree in Computer Science from the University of Stuttgart, Germany, in 1988. He then joined IBM Research to work on natural language technology. In 1995, he moved on to IBM Development, working on information retrieval, text mining, and unstructured information management. In 2002/03, he represented IBM in the W3C workgroup to define an XQuery Fulltext standard. Since 2004, he is the lead architect for Linux^{TM} on Cell development at the IBM Böblingen lab.

Jan **van Lunteren**
IBM Corporation [http://www.ibm.com]
IBM Research GmbH – Zurich Research Laboratory [http://www.zurich.ibm.com/]
Säumerstrasse 4
Rüschlikon
8803
Switzerland
jvl@zurich.ibm.com

Jan van Lunteren received the M.Sc. degree in Electrical Engineering, the M.Sc. degree in Technological Design, and the Ph.D. degree in Electrical Engineering in 1992, 1994, and 1998, respectively, all from the Technical University of Eindhoven, The Netherlands. He has been with the IBM Zurich Research Laboratory, Rüschlikon, Switzerland, since 1994, doing research on high-speed networking. His current interests include high-performance memory systems, (deep) packet classification algorithms, and hardware-based XML acceleration.

Paul **Herrmann**
Universität Leipzig - Institut für Informatik [http://www.informatik.uni-leipzig.de/]
Augustusplatz 10-11
Leipzig
04103
Germany
paul@informatik.uni-leipzig.de

Paul Herrmann received the diploma degree in Physics in 1967 and the Ph.D. degree in Physics in 1973, both from the University of Leipzig, Germany. Until 1989, he worked at the university's electronic data processing center. Since then, he is with the Department of Computer Science of the University of Leipzig, focusing on electronic design automation, client-server architectures, and z/OS (OS/390).