Linux TCP Network Stack Analysis and Partitioning for Network Processors

Ralf Lehmann and Alexander Schill
Institute for System Architecture, Chair for Computer Networks
Technische Universität Dresden
01062 Dresden, Germany
email: {lehmann, schill}@rn.inf.tu-dresden.de

ABSTRACT
Current high speed networks cannot be fully utilized by today’s high end systems. The processing requirements of applications and advanced services or security processing aggravate this situation. To overcome these shortcomings intelligent network cards with network protocol offload engines, for example based on a network processor, are necessary. To reduce development time, we reuse existing software protocol stack implementations for partitioning and implementing on the network card. Since manual partitioning is very expensive and fault-prone, we use a modified automatic debugging tool for the analysis. We outline the results of Linux TCP network stack analysis and partitioning when using an automatic debugger.

KEY WORDS

1 Introduction and motivation
In the past the available network bandwidth especially in local area networks has grown much faster than the computing power of the connected high end computers. Thus even such high end computers cannot fully utilize local high speed networks [6]. Most of the current used network transmission protocols like HTTP, FTP and SMTP are based on TCP/IP. However this protocol was not designed for todays high speed network transmissions using local networks with a low bit error rate. Hence the protocol computing overhead is primarily responsible for the data transmission obstacle. Due to the wide and heterogeneous usage of protocols like TCP/IP any solution has to be standard conform without any modification of the protocol.

To overcome this bottleneck of protocol computing so-called protocol offload engines were developed. For mass production asic designs are preferred. Though such plain hardware based designs have development cycles of more than a year and are inflexible when protocol enhancements or changes have to be implemented. Another solution for the realization of a protocol offload engine is to use special IO optimized processors, so called network processors which are specialized in processing network packets but still freely programmable, like the Intel IXP [4] network processor family.
An approach for a TCP protocol processing acceleration is described in [1]. Main goal of this concept is to partition an existing software implementation of a network protocol stack into signalling and data transmission parts and implement the time and CPU consuming data transmission part of the stack onto an IXP network processor. However the main problem of this idea is to specify all the needed synchronization data between the plain software part of the implementation still running on the host processor and the extracted fast data path running on the network processor.

To overcome this problem we present in this paper an approach for partitioning of the data path of the Linux [9] TCP protocol stack and specify the needed synchronization data for an implementation.

In the following chapter we discuss possibilities for the partitioning process and the involved problems. Thereafter we analyze the TCP network stack implementation of the Linux kernel. Next the analysis results are presented and interpreted. After classifying our approach in comparison to related work, we finally conclude our results and address future work.

2 Partitioning and acceleration approach

Independently which network protocol stack has to be analyzed, the CPU and time consuming parts of the implementation responsible for high speed data transmission have to be localized. Therefore especially if the network processor implementation should be based on an application with a huge amount of source code like the Linux kernel, the network stack should always be analyzed with profiling tools [6, 7] in the first instance.

As a result of the real processing measurement in [6] only a small amount of source code of the original network stack implementation has to be extracted to follow the approach in [1] – the so-called fast path of the data transmission without error handling but error detection.

Plain manual partitioning of more than 10,000 lines of source code like the receive part of the TCP network stack implementation is too expensive and fault-prone, especially for source code which is not well documented. Therefore a software analysis approach described in [8] was used. With this analysis tool implemented on top of the GNU debugger gdb [3] it is possible to log all used code lines with all referenced data structures. Based on these log files the extraction of the TCP fast path and the declaration of the synchronization data will be possible.

3 TCP network stack analysis


Since no time or CPU specific measurement was done, a simple loopback connection could be used for tracing and analyzing the kernel functions responsible for a TCP connection.

The protocol stack source code tracing and analysis was done using a modified automatic GDB [3] debugger presented in [8]. The modifications of the automatic debugging tool were necessary to get User-Mode-Linux compatibility for the debugging tool. Now the tool is able to connect to a running kernel process.

At first, only the receive path of the TCP network stack was analyzed.

For tracing and analyzing of the receive path of the Linux TCP network stack, the initial breakpoint was set at ip_rcv, since the device layer and the interface layer depend on the used hardware. Furthermore an implementation of the ethernet and routing functionality are part of an existing module library for the Intel IXP network processor.
Figure 1 illustrates the analyzed parts of the receive path of the network stack. Since the input queue of the TCP layer is skipped if the TCP layer is idle, both layers were processed in one run. Due to the used heuristic method not only one received TCP packet was traced but a whole TCP connection.

4 Analysis results

4.1 Source code analysis

After completing the trace debugging all needed source code for partitioning the TCP receive path is logged and therefore known. Figure 2 illustrates all needed function and sub-function calls for a complete trace of a received TCP network packet. Furthermore due to the logging of the executed source code all source code parts responsible for error handling and therefore no candidate for the implementation on the network processor are skipped. Thus the amount of source code to be extracted was reduced to less than 10% of the whole source code of the TCP protocol stack.

4.2 Data structure analysis

Network processors like the Intel IXP have different memory buses for each type of available memory type on the network card plus a few local registers depending on the type of the network processor. Since all these memory types differ in terms of price, latency and throughput it is necessary to have a list of all needed data structures and the frequency of their read respectively write access for optimal usage.

Figure 3 illustrates all the accessed data structures when tracing the receive path of the TCP stack. The data structures are separated by their memory address or – if a processor register was used – their name in conjunction with the used register.
4.3 Synchronization data analysis

For getting all needed data for synchronization between host system implementation and network processor implementation all the data read and write accesses and the time/trace sequence of access have to be analyzed. If a data structure was read but not written before, it has to be copied from the host system during the synchronization process.

4.4 Usage of analysis results

The main usage of these results are to facilitate the partitioning decision of the Linux TCP network stack implementation and implementing it on the network processor. To get information about parallelizing the extracted source code it is furthermore necessary to find the conflicting read and write accesses of all used data structures – as a result of the trace debugging too – and separate all independent code parts for implementation on the special micro engines of the network processor.

5 Related work

An approach for a TCP acceleration based on network processors is described in [1] but only with manual partitioning. A comparable approach but for IPsec acceleration is described in [7]. Another analysis for the TCP protocol is presented in [5]. All these approaches follow only the analysis of a special network protocol with manual optimization needs.
Figure 3: TCP receive path – Data structure usage
6 Conclusions and future work

For easing the burden of network protocol processing on today’s host processors partitioning approaches with usage of protocol offload engines are useful. However partitioning an existing software implementation of a protocol stack is too complex for manual procedure. Using an automatic debugger for data collection is an acceptable approach for network stack analysis as well as other source code analysis. The traced data about data structure access and code usage are an optimal basis for protocol stack partitioning on microengines of a network processor. Thus the amount of source code of the Linux TCP network protocol stack for partitioning can be reduced to less than 10% of the whole source code. Furthermore an optimal implementation on a network processor in consideration on memory accesses and parallelizing is possible.

Next steps for partitioning are analyzing the send path of the Linux TCP network stack and detailed parallelizing of the partitioned code as well as analyzing other network protocol stacks like an IPsec protocol implementation.

References


