An Encryption-Enabled Network Protocol Accelerator

Steffen Peter, Mario Zessack, Frank Vater, and Michael Methfessel

IHP GmbH, Im Technologiepark 25, 15236 Frankfurt/Oder, Germany
{peter,zessack,vater,methfessel}@ihp-microelectronics.com

Abstract. Even in light-weight wireless computing applications, processing of network-protocols becomes more and more computation- and energy-hungry, with increasing data rated and the need for security operations. To cope with such requirements and as alternative to heavy-weight computation systems we propose an embedded system that is build for fast network-processing while supporting acceleration of state-of-the-art symmetric (AES) and asymmetric (ECC) cryptographic operations. We demonstrate how to build a dedicated TCP accelerating system based on a profiling analysis. We also discuss optimized implementations of the AES and ECC cryptographic protocols while considering the trade-off between software and hardware. Compared to an initial software-only implementation our final system accelerates the protocol handling by a factor of three, while the cryptographic operations are improved by two orders of magnitude. Our system which was manufactured in 0.25μm CMOS technology needs about 55 mW for a data rate of 40 MBit/sec.

1 Motivation

Light-weight networked devices as they are applied in the emerging world of ubiquitous computing have to cope with an increasing amount of data. For instance, surveillance applications range from small sensor readings up to real time video. Handling this data requires reliable, fast, and efficient network processing. Additionally, wireless and embedded devices are more and more becoming integral parts of safety-critical and long-living systems, e.g., as components of applications which monitor buildings, cars etc. This implies that strong security means must be employed in order to ensure data integrity and authenticity. Suitable encryption methods are generally very demanding computationally. Together, there is a need for efficient network processing in combination with state-of-the-art encryption methods. In this paper, we perform hardware/software co-design to develop and implement a network processor with encryption capabilities.

Such a processor is useful in the context of computers with limited resources, such as laptops or PDAs. Another application is to enable "dumb" devices with secure network access. Further applications are in the area wireless sensor networks (WSN). Sensor nodes usually cannot easily bridge from their internal network protocol to wide area networks. Such gateway tasks are typically considered to be performed by heavy-weight personal computers (PC). This indeed solves the technical problems, but economically and practically this approach is often not viable. Consider the scenario of border land protection: the environment-observing nodes transfer their results to a cluster head which forwards the results to a control center. Such cluster-head nodes are required every 50 to 100 meters, making PCs unfeasible.

An additional task performed by the cluster head is security control. Although an abundance of security protocols for wireless sensor networks have been proposed, the most promising solutions require a trust center. The tasks of such trust center include observation of the network behavior (intrusion detection), cryptographic operations, and authentication. The battery and processing power of embedded devices is often not sufficient to run the required strong security algorithms. This underlines the need to equip light-weight devices with hardware accelerators, in order to increase the performance and reduce the energy consumption for cryptographic operations in combination with network processing.

In this paper we propose an integrated solution, suitable for the described applications. Although we focus on the TCP/IP protocol, the results are applicable to other transport protocols as well. The solution includes cryptographic hardware accelerators that sufficiently improve the applicability of strong secure cryptographic algorithms in such environments. Though the discussed design is a concrete implementation, the paper provides a general blueprint for light-weight, hardware-accelerated implementations which combine network protocol handling and cryptographic operations.

The rest of this paper is structured as follows. In Section 2 we profile a TCP implementation in order to determine the performance-critical operations. Then we evaluate potential solutions for the bottlenecks while referring to related in work in that area. Based on these results, we describe a general network accelerator design in Section 4. Potential solutions for fast cryptographic implementations are investigated in Section 5. In Sections 6 and 7 we discuss the implementation and the results before we conclude the paper with a short summary.

2 Profiling TCP on an Embedded Processor

A first step toward an efficient implementation is to understand what the critical and time-consuming tasks are. For the TCP protocol, a comprehensive performance analysis has already been reported in [5]. The focus there is on computer systems with the Windows or Linux operating system. The results show that about 50% of the processing effort is kernel or driver-related. In single-thread-operating systems, as they are applied in embedded environments, we expect significantly different figures. For a reliable determination of these values we implemented a TCP/IP stack on an embedded system. We added hooks that allow us to measure the time spent in the various subroutines.

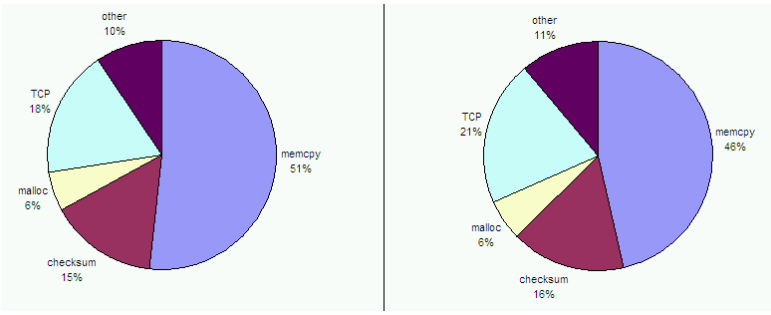

Fig. 1: Profiling results for transmitting (left) and receiving (right)

The implementation includes a fast path that processes incoming and outgoing data much faster when no special treatment is needed. That is, if the data packets arrive in correct order without errors, there is no need to process the full conditional TCP logic. In our test cases, 5 Megabytes of data were transferred. We assumed a good connection that only passes the slow path twice per connection, namely to establish and to close the connection. The 1815 data packets do not enter the slow path but are entirely processed in the fast path. Assuming reasonably good transmission conditions in realistic cases, the obtained values show where to invest effort in order to improve the efficiency.

As a first result, the total processing time indicates that receiving needs slightly more effort than transmission (4.7 sec. for send, 4.9 sec. to receive). This is not particularly surprising and has already been reported in [1]. The more relevant differences are in the distribution of processing time among different processing steps. An overview of the values for sending and receiving can be seen in Figure 1.

During transmission, most time is spent for copying data from the application memory to the TCP send buffer and the packet structure. Computation of the checksum and the final send operation also require significant processing time. During reception, the major operations are copying of the data from the network memory to the internal TCP structure, computation of the checksum and finally transfer of the data from the TCP memory to the application. On the other hand, the actual logic of the TCP protocol (the state machine, congestion control, and the conditional logic) does not need even 20% of the processing time.

3 Related Work

The profiling results published in [5] already indicate that the actual protocol processing and the checksum computation are not the major performance bottlenecks of a TCP implementation. Basically our results confirm this. Nevertheless, in work related to our effort, the two most discussed approaches to accelerate a TCP implementation by dedicated hardware propose acceleration of these functions. The first approach of such Offload-Engines [6] is to design the TCP state machine in hardware. However, with a software-implemented fast path for general data sending and receiving, the complicated state machine is bypassed most of the time. The relevant processing time we estimate is merely 3%. Thus the potential advantages of a protocol state machine in hardware are minor. In contrast the potential disadvantages of a hard-coded protocol implementation - in particular the lack of flexibility - would clearly outweigh the benefits.

The second well-known approach is the implementation of the checksum operation in hardware. A hardware block to compute the TCP checksum is very small (a 16 bit adder and a register) so that the silicon costs are negligible. However, our profiling results show that the potential performance gain, while not negligible, is not of primary importance. The checksum operation for both receiving and sending requires merely 16% of the total processing time.

An alternative to Offload engines for TCP are Onload implementations that implement the packet processing onto a dedicated set of computer resources. Usually it means to dedicate a processor core. According to [10] that approach allows optimizations that are otherwise impossible when time sharing the same compute resources with the operating system and applications. Though the idea of an dedicated CPU core for network processing is interesting it does not suit in our light-weight scenario.

Several Offload and Onload engines as well as hybrid approaches [13] have been proposed. However, most of them focus on TCP server applications which have - as our profiling results already showed - different requirements and properties. Anyway they do not solve the actual bottleneck we could determine in the profiling. The most time consuming operation is copying. Since our tested implementation requires two copy operations per direction, the problem becomes even more apparent. The additional copy operation from and to the network adapter cannot be omitted. Single-copy or even zero-copy TCP stack implementations have been proposed to tackle the problem [14], but this typically compromises the socket API and requires undesirable non-standard handling in the user code.

An unexpected result of our profiling is that memory allocation does not require a notable amount of processing time: allocation consumes about 6% of the total time. Each packet needs a memory slot and has to be stored in retransmit or receive buffers. On personal computers, memory management is a heavily operating-system-supported task. Since we do not have such a unit on our embedded device this task must be considered.

4 System Design

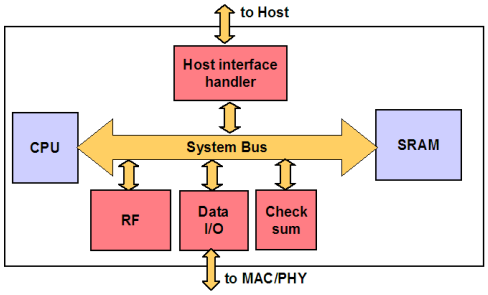

Fig. 2: General design of our network accelerator: Checksum can be computed while data is written into memory.

Based on the results of the profiling, we can design a TCP offload device with the primary targets of reduced energy consumption and support of a data rate of 54 Mbit/sec, without yet considering encryption. As goal for the power consumption, a value of 50 mW was targeted for an implementation using the in-house $0.25\mu m$ CMOS Technology. In order to archive these goals the main strategy is to reduce the CPU load by doing specific steps in hardware: a) copy operations, and b) evaluation of the checksum. Other parts of the TCP protocol are done in software on the embedded CPU. With this general strategy, we expect to reduce the utilization time of the CPU by more than 80% (in comparision to a reference solution which simply ports the protocol to the embedded processor) because only the protocol handling (<20%), memory allocation (5%) and additional control information have to be processed by the CPU.

Our basic architecture is shown in Figure 2. The offload device consists of these components:

CPU: The CPU is a 32 bit MIPS processor.

Bus: The components are connected by a 32 bit AMBA bus [8].
SRAM: Packets are stored in a 32 kByte SRAM memory.
RF: A register file controls the operation of the system.
Checksum: The checksum is computed for the data transferred over the bus
Network device interface handler: The hardware unit copying data between the internal SRAM and the network device.
Host interface handler: A unit connecting to a host device

The design does not contain a separate memory allocation unit. Although the memory operations consume a noticeable slice of the total CPU time, we decided not to implement a dedicated memory manager hardware block. Tests showed that such a unit would accelerate the memory allocation by 77%, even considering the additional system communication overhead. However the the additional hardware costs (300 flip-flops for the 32kB SRAM) and the potential lack of flexibility deterred us from exercising this option.

Now consider the case that the host wants to send data. First, it would register the transmission, so that the embedded CPU would create a socket and allocate a memory area. The host would directly copy the data into the assigned memory region, whereby the checksum is computed. Whenever a packet is full, the CPU finishes the header processing. Then the network device is given the address of the packet and finally transmits it on its own.

Incoming packets will also be copied to an assigned memory area in the internal SRAM. During this copy operation the checksum is computed. If the incoming packet is received and not corrupted, the CPU is informed and starts to process the header. At all other times the CPU can sleep and thus reduce the energy consumption.

This design employs the CPU solely for complicated operations which are not of primary significance for the total effort, i.e.,

- Build-up and tear-down of a connection.
- Management of the state machine and sockets.
- Congestion control: adaption of the transmit rate to network load.
- Error handling: unexpected packets, retransmission etc.
- Software handling allows protocol variations and debugging.

One sees that the CPU is not involved in those steps which touch the data payload (copying and checksum). It operates only on packet headers and on internal data needed for book keeping. Since the amount of this data is small compared to typical packet payloads, the implementation is expected to be efficient.

As described, the system is driven by an external host, allowing the CPU to sleep a large percentage of the time. Alternatively, the system could be used as a stand-alone solution, as would be the case for a light-weight gateway in a WSN. Here the CPU would process the application code in the remaining time. In either case, the CPU could also be used for cryptographic operations, e.g. to encrypt the payload or to handle cryptographic protocols. In the next section, we consider how to add encryption in hardware to the design.

5 Cryptographic Functions

In this section we discuss implementations for two prototypical cryptographic protocols: AES and ECC. Both are standardized and considered as strongly secure.

Since our goals are good performance and reduced energy consumption, we will discuss hardware accelerators and their potential trade-offs.

The Advanced Encryption Standard (AES) is the replacement for the insecure DES-Algorithm. It was standardized in 2001 [16] by the NIST. It is a symmetric cipher algorithm, which uses the same key for encryption and decryption. The data block length is 128 bit. The data is arranged in a matrix of 4x4 bytes. In contrast to the data block length, various key lengths are possible. Three different key lengths are standardized. The shortest, and also the most often implemented version, has a length of 128 bit. Longer (but rarely used) key lengths are 192 and 256 bit. Especially for low area implementation those versions are not well suited. They require additional area for key registers and the runtime increases by up to 30%.

The AES algorithm itself consists of four parts in 10 rounds: key addition, byte-wise substitution, shift in rows and mixing of the columns. The nature of every step allows an efficient implementation in hard- as well as in software.

The implementation applied in our system is similar to [17]. The standard implementation has a memory-like interface with a 4 bit address and a 32 bit data bus. First, the key is loaded into the key register. Next, the data to be encrypted/decrypted is loaded. The algorithm starts automatically after the last chunk of the four 32 bit data blocks is written. We have adapted the key management so that it is possible to select between two different stored keys. We used a full 128 bit wide key interface instead of the memory-like interface. Changing the key (e.g., to use different keys for encryption and decryption or for bidirectional communication) costs only a control word instead of a control word plus four times a data word (32 bit).

To optimize the implementation, we analyzed the requirements and the performance of a possible straight-forward AES implementation. We share the S-Box with the key generator and the algorithm itself Fig. 3 b). Furthermore we replaced the standard Mixcolumn function. Usually an independent component is used for every direction an own. We reuse the Mixcolumn function for encryption in the decryption path as mentioned in [REF]. As result we save app. 44% of silicon area.

As alternative software implementation on our embedded CPU we simulated encryption and decryption with an optimized AES software implementation[2]. Table 1 shows the measured results for both software and hardware. The latter - independent of direction - requires 78 clock cycles including I/O-Operations. Assuming that the MIPS core requires 44mW at 33MHz a 1MByte block consumes 95.3mJ in power to encrypt the data. The AES core requires for the same data block 1.04mJ. Furthermore the hardware accelerator is about 67 times faster than the encryption in software and 89 times faster than decryption in software.

Table 1: Comparison of the hardware AES design to a MIPS software solution

		Clock cycles	Time [μs]	Energy per 1MByte [mJ]
En-	Hardware	78	2.5	1.04
cryption	Software	5228	172.5	95.30
De-	Hardware	78	2.5	1.04
cryption	Software	6857	226.3	125.00

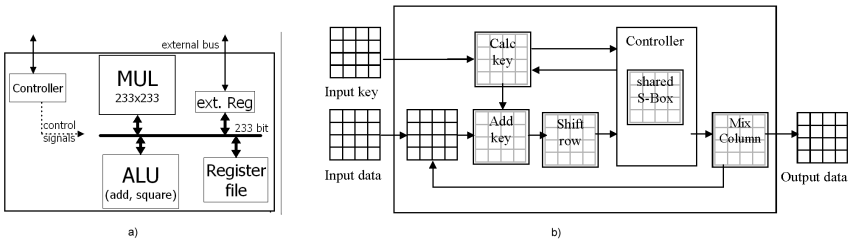

Fig. 3: Schematic of a) the ECC design and b) the AES accelerator

5.1 ECC

Symmetric cryptographic approaches, like AES, are considered to be secure and computation costs are relatively low. However they do not always provide satisfying answers to questions regarding authentication, key distribution, and ensuring of data integrity. Here asymmetric approaches, also known as Public Key Cryptography (PKC), are a suitable solution. We focus on Elliptic Curve Cryptography (ECC) since it provides a good level of security even with relatively short key sizes, leading to relatively low calculation costs compared to other PKC-approaches. But, we are convinced that processing time and power consumption are still too high if all operations are executed in software on a light-weight device. This is why we propose a hardware design that can accelerate the ECC operations on our network processor.

Figure 3 a) depicts the block diagram of the exemplary 233 bit ECC hardware accelerator. The arithmetic units and register are connected by a 233 bit internal bus. The control unit manages the bus access and the operations. This is the place where the ECC algorithms are executed. In our design the elliptic curve point multiplication (ECPM) is performed by the Lopez-Dahab algorithm [9]. The control unit also manages the access to the eight 233 bit registers. One of these registers can additionally be written from the external bus.

The ALU combines the functionalities of addition, squaring and allows bit manipulations. The multiplier is an Iterative Karatsuba Multiplier, as proposed in [4]. It requires 9 cycles for each 233 bit operation. It is not only the largest unit but also the most utilized one. The duty time is more than 90%.

As for the AES we want to compare the hardware design to a software implementation on the embedded CPU executing the ECC operations. The ECC software implementation is based on the MIRACL library [12] and was run on the system without utilizing the crypto-accelerators. A point multiplication on the curve B-233 takes 13 million clock cycles which corresponds to 400 ms. The code size for this implementation is 48 kilobytes. An alternative implementation requires only 14 kilobytes but is much slower with 900 ms required for a point multiplication. The code size must be considered when memory is an issue, as it is for many small mobile devices.

Table 2 shows a comparison of the 233 bit MIRACL software implementation with the ECC hardware design as both are implemented and running on the communication SoC. All the data were measured in the simulation environment for one 233 bit point multiplication at a speed of 33MHz. We used the simulation environment in order to isolate the power consumption for the operation. The results were verified on the actual hardware after manufacturing the chip.

Table 2: Comparison of the 233 bit ECC hardware design to a software solution on the SoC

	Time[ms]	Power[mW]	Energy[mWs]
software	410.2	40.2	16.490
hardware	0.4	75.6	0.030

The results show that the hardware solution is 1000 times faster and consumes 550 times less energy in comparison to the software implementation.

It should be mentioned that efficient assembler language supported software implementations can give better performance than the chosen MIRACL library. For example the StrongARM (206 MHz) implementation presented in [11] needs 9 ms and less than 4 mWs. It still is significantly slower than the hardware design and needs two orders of magnitude more energy per ECC-operation.

5.2 Integration of the Crypto-Accelerators in the SoC

We decided to integrate both cryptographic accelerators into the network accelerator. The cipher AES is suitable to protect the transmitted payload. The more expensive public key approach ECC is used for key establishment and for authentication, but is not intended to encrypt the payload.

The two different areas of application have an impact on our integration decision. The AES module is located before the internal SRAM. When enabled, it transparently encrypts (or decrypts) the data flowing into or out of the packet memory. In this way, the initial copy operation can fill the packet payload, compute the checksum and encrypt the payload. For a correct processing of the data on a receiving peer computer, it is important that the checksum is computed over the already encrypted data. in any case, all these operations are executed transparently within one actual copy operation.

Since ECC is not used to encrypt payload data, we decided to connect it directly to the system bus. The CPU (but also an external host) can address the registers of the ECC accelerator and transmit data and keys. While the ECC unit is working, the CPU can perform other tasks or sleep.

6 Implementation of the TCP/Encryption chip

Before starting with the concrete discussion of the system design, the general work flow is described.

All functional blocks (excepting the MIPS core, the AMBA bus, and SRAM) were implemented in VHDL. Behavioral simulations of the individual VHDL blocks verified the correctness of the implementations. Comprehensive hardware/software co-simulations of the complete system with the embedded TCP stack could finally show the correctness of the entire design and could provide first estimates of the performance and energy consumption. In addition to the simulations, we ported the design to an FPGA in order to test the system in the the real world at real speed.

Our primary target was an implementation in silicon. The ASIC implementations were synthesized with the library of our in-house $0.25\mu m$ CMOS Technology[7]. The resulting netlists with detailed timing informations were the basis for the calculation of the power consumption using Synopsys PrimePower[15]. PrimePower

provides a gate level power estimate based on the parameters of the technology library and realistic test pattern. It results in very precise power estimate, since the real transitions for the calculation are considered instead of merely statistical assumptions.

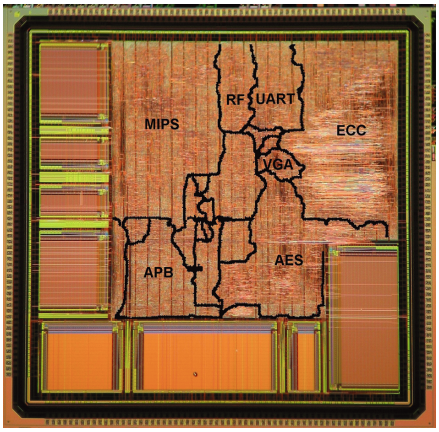

Fig. 4: The final system on chip. We also marked the largest internal blocks. APB is the bus and RF is the central register file. The rectangle blocks are memories.

Since it is our goal to test the concept of the chip in as many environments as possible, we decided to add several general purpose inout/output interfaces (UART, GPIO, SPI). The connection to the host computer is via a CardBus interface. The connection to the network device is either per UART or EPP interface. The current prototype requires a rather large number of pins (256, of which 219 are signal pins). To a large part, this number arises from the CardBus interface and the 32 bit wide bus to the external program memory.

A chip photo can be seen as Figure 4. The pads and the internal SRAM and cache (total 56 kByte) determine the total core size (54mm^2). A dedicated application specific design would not need all components and interfaces, and thus reduce the total area significantly. Also, a state-of-the-art 0.13 μm CMOS technology would additionally reduce the size for the chip.

For the results discussed in the next section, the open-source lwIP TCP/IP protocol stack [3] was ported to our system, extended by memory management functions malloc and free in software. No operating system was used. The CPU executes one task at a time, and goes into a power-saving sleep mode when done. The sleep mode is left when an interrupt is given by the register file, which controls the overall system.

7 Results and discussion

Our goal was to build a network accelerator that reduces power consumption and increases the network performance of an embedded system on chip. Our approach was to implement dedicated hardware accelerator blocks while the complicated protocol operations are still performed in software on the embedded processor. In this section we want to answer the main questions: what we actually have gained by the

hardware support, and whether the original goals were met. For this purpose we estimated (1) power consumption, and (2) attained data rate for three cases:

1. (SW) Pure software implementation on the embedded MIPS processor, at maximal data rate attainable.
2. (HW 1) With hardware accelerators, at the same data rate as (SW).
3. (HW 2) With hardware accelerators, at the maximal data rate attainable.

All values correspond to a clock frequency of 33.3 MHz, which is the clock supplied by the CardBus interface.

Table 3: Results of performance and power consumption of three scenarios.

Case	Rate (Mb/sec)	CPU active	CPU (mW)	Bus (mW)	Regs (mW)	I/O (mW)	Total power
SW	20.7	100%	60	14	7	8	89 mW
HW 1	20.7	15%	9	14	7	12	42 mW
HW 2	40.0	31%	18	14	7	16	55 mW

Table 3 splits the consumed power by the different system blocks, based on the PrimePower simulations. Measurement of the total consumed power for the actual chip gives somewhat larger values. One reason is that the power consumed by the pads is also included in the measurement. The table shows that the hardware design indeed increases the attainable data rate and reduces the energy consumption significantly. The maximal data rate in pure software on the MIPS CPU is 20.7 Mbit/sec. In this case the CPU is utilized for 100% of the time, by construction. Here the CPU (including cache) requires about 70% of the total power (60 mW). We see that the target data rate of 54 Mbit/sec can not be reached by the pure software version.

The second configuration (HW 1) runs at the same speed as is attainable by the software implementation. By comparing these cases, we can identify the power saving due to the hardware blocks. The CPU is now utilized for only 15% of the time. Thus, the hardware accelerators reduce load on the CPU and thereby save more than 50% of the total power. However, the factor is not quite as large as the 80% expected from the profiling. Indeed, the power consumed by the CPU alone does decrease by 85%, offset by a small increase due to the hardware I/O. It is the overhead due to other blocks such as the system bus, register file, and I/O which becomes a significant portion in case HW1. The power consumption of I/O, register and bus are roughly at the same level as the CPU. this indicates that further optimizations are not as easy since no single block needs more than 15% of the total energy.

The third row shows the data rate and power consumption for the maximal data rate attainable with the implemented system. With 40 Mbit per second it is lies below the target of 54 Mbit/sec. Our investigations resulted in the conclusion that the current Cardbus interface to the host does not allow a higher data rate (burst mode is not yet supported). The system design itself would allow more than 100 Mbit per second at 33 MHz clock frequency. At the maximum data rate for the manufactured system of 40 Mbit/sec, the CPU is busy for 31% of the time. The other 70% of the CPU time could be used for an embedded application.

For the encryption blocks, the performance was verified for the manufactured system. From the PrimePower analysis, the consumed power for the AES block is

17 mW when idle and 52 mW when active. For the ECC unit, the idle power is 25 mW and the active power is 60 mW. For simplicity, these contributions were omitted in the discussion of the power consumption for protocol handling above. The relatively high idle values for the encryption units shows the necessity of using means such as clock gating to eliminate the power used by idle parts of the design.

Overall, results for the TCP protocol processor are close to the originally targeted values, but do not quite reach them. The gain by hardware accelerators for the protocol processing is not as spectacular compared to the accelerator blocks of the cryptographic operation, where we could improve the results by two orders of magnitude. For protocol processing the advantage factor of the network acceleration for performance and power consumption are five and two, respectively. Of course, the power consumption would be reduced further if the same design is implemented in a more advanced technology with smaller gate length.

8 Conclusions

This paper describes the design and implementation of an offload protocol processor which can efficiently handle TCP protocol processing together with encryption by AES or ECC in hardware. Based on profiling of a software TCP implementation, copying of data payloads and (to a lesser extend) evaluation of the TCP checksum were identified as suitable candidates for hardware acceleration. These are the processing steps which pass over the data payload. Operations on the packet headers and for TCP bookkeeping consume only a small percentage of the total power. These are therefore best done in software on an embedded CPU, permitting more flexibility and avoiding the development effort for dedicated hardware units. The system designed along these lines was further enhanced by adding hardware blocks to perform both AES and ECC encryption. Since AES was intended for encrypting the data payloads, the AES unit was placed in the data flow before the internal SRAM. The ECC unit was treated as a distinct independent entity, which communicates with the rest of the system via registers. This allows the lengthy ECC en- or decryption to take place in parallel to other operations on the CPU. The final design was manufactured using the IHP in-house 0.25μ CMOS technology.

A combination of measurements on the manufactured chips and detailed simulations for the power consumption was done to determine the performance of the system and to split the consumed power by the different design units. As a basis for comparison, the TCP protocol was done completely in software on the embedded CPU. In this case, a data rate of 20.7 Mbit/second can be reached at a power consumption of 89 mW, whereby the CPU is running continuously. If the hardware accelerators are used for the same data rate, the utilization of the CPU drops to 15% and the power to 42 mW. A large part of the power now is consumed by the system bus, the register file, and I/O. The maximal data rate which can be attained with the system is 40 Mbit/sec at a power consumption of 55 mW. Here, the CPU is still only active 31% of the time, since the rate is limited by the used CardBus interface to the host.

The results indicate a number measures which could be implemented in an improved design. First, better solutions should be sought for units such as the system bus or register file, since these dominate the power consumption when the TCP hardware accelerators are used. Second, a more efficient way to move data between the host and the system such as DMA should be used. Third, clock gating should

eliminate the power consumption by blocks which are idle, notably the encryption units, when these are not actively used.

Overall, the presented system is an efficient implementation of a combined protocol/encryption processor, which could be further improved by measures suggested by the evaluation of the manufactured system.

References

1. D.D. Clark, V. Jacobson, J. Rornkey, and H. Salwen. An analysis of tcp processing overhead. In *IEEE Communications Magazine*, pages 23–29, 1989.
2. Joan Daemon. *AES implementation, optimized ANSI C v2.0.* available at http://www.iaik.tugraz.at/research/krypto/AES/old/ rijmen/rijndael.
3. Adam Dunkels. *lwIP – a lightweight TCP/IP stack.*, oct 2002. http://www.sics.se/ adam/lwip/.
4. Zoya Dyka and Peter Langendoerfer. Area efficient hardware implementation of elliptic curve cryptography by iteratively applying karatsuba's method. In *DATE*, pages 70–75, 2005.
5. A. P. Foong, T. R. Huff, H. H. Hum, J. R. Patwardhan, and G. J. Regnier. Tcp performance re-visited. In *ISPASS '03: Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 70–79, Washington, DC, USA, 2003. IEEE Computer Society.
6. Doug Freimuth, Elbert Hu, Jason LaVoie, Ronald Mraz, Erich Nahum, Prashant Pradhan, and John Tracey. Server network scalability and tcp offload. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, pages 15–15, Berkeley, CA, USA, 2005. USENIX Association.
7. Innovations for High Performance microelectronics. *IHP microelectronics: technology*, 2006. http://www.ihp-ffo.de/24.0.html.
8. ARM Limited. AMBA specification, revision 2.0, 1999. Available from ARM website http://www.arm.com.
9. Julio López and Ricardo Dahab. Fast multiplication on elliptic curves over GF(2^m) without precomputation. In *CHES '99: Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, pages 316–327, London, UK, 1999. Springer-Verlag.
10. Greg Regnier, Davev Minturn, Gary McAlpine, Vikram Saletore, and Annie Foong. Eta: Experience with an intel xeon processor as a packet processing engine. In *11th Symposium on High Performance Interconnects*, Los Alamitos, CA, USA, 2003. IEEE Computer Society.
11. Ingo Riedel. Security in ad-hoc networks: Protocols and elliptic curve cryptography on an embedded platform. Master's thesis, Ruhr-Universitaet Bochum, 2003.
12. Michael Scott. *MIRACL—A Multiprecision Integer and Rational Arithmetic C/C++ Library, Version 5.0.* Shamus Software Ltd, Dublin, Ireland, 2005. Available at http://indigo.ie/ mscott.
13. Leah Shalev, Vadim Makhervaks, Zorik Machulsky, Giora Biran, Julian Satran, Muli Ben-Yehuda, and Ilan Shimony. Loosely coupled tcp acceleration architecture. In *HOTI '06: Proceedings of the 14th IEEE Symposium on High-Performance Interconnects*, pages 3–8, Washington, DC, USA, 2006. IEEE Computer Society.
14. Peter Steenkiste. Design, implementation, and evaluation of a single-copy protocol stack. *Software Practice and Experience*, 28(7):749–772, 1998.
15. Synopsys Inc. *PrimePower: Full-Chip Dynamic Power Analysis for Multimillion-Gate Designs*, 2005. http://www.synopsys.com/products/power/primepower_ds.pdf.
16. FIPS U.S. Department of Commerce/NIST. Advanced Encryption Standard (AES), FIPS PUB 197, 2001. Available from http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.
17. Frank Vater and Peter Langendörfer. An area efficient realisation of aes for wireless devices. *it - Information Technology*, 49(3):188–, 2007.