Comparison of FPGA-based Implementation Alternatives for Complex Algorithms in Networked Embedded Systems – the Encryption Example

Enrico Heinrich, Sebastian Staamann, Ralf Joost, Ralf Salomon
Institute of Applied Microelectronics and Computer Engineering
University of Rostock, 18051 Rostock, Germany
{enrico.heinrich,sebastian.staamann,ralf.joost,ralf.salomon}@uni-rostock.de

Abstract

Field-programmable gate arrays provide a flexible and easy-to-configure implementation platform that supports the development of tamper-proof networked embedded systems. Many of these systems employ the Advanced Encryption Standard algorithm in order to achieve a secure mode of operation. Since this algorithm is of a high computational complexity, this paper utilizes various hardware-software co-design techniques for its implementation. These techniques vary in the degree of required design expertise, the degree of how (software) functionalities are implemented in hardware logic, and the achievable speedup, which is about 12 to approximately four thousand.

1. INTRODUCTION

The general view on processing systems is that they consist of two different layers, hardware and software, which have different properties with respect to speed and flexibility. Everyone "knows" that hardware is a piece of silicon that has undergone a long design and manufacturing process. Designing a particular hardware involves the logic design, wiring and routing, as well as the production of the photo masks. It thus does not come to a surprise that the development of Nvidia's state-of-the-art Smartphone processor APX 2500, for example, took more than 800 men-years. Therefore, hardware is a rather static layer. As a consequence, hardware designers aim to provide general-purpose designs that are as fast as possible.

Software, on the other hand, is much slower but much more flexible, and realizes the required functionality. Any modification can be easily done in an editor, and after re-compilation, the software provides the new functionality.

The traditional combination of hard- and software works very well in most applications. From a security perspective, however, software is problematic, since it can be quite easily altered (compromised) after its deployment. Backdoors, viruses, worms, and Trojan horses are typical, well-known examples for modification attacks. In order to make sure that such attacks do not cause any damage, it is common practice [15] to install some non-modifiable, tamper-proof security control hardware between the network and the actual application host.

Due to the advancements in the design of digital systems, networked *embedded* (real-time) systems have received recent attention. Throughout this paper, the term "embedded system" refers to small electronic devices that are tightly integrated into larger systems that are interacting with their environments. In order to realize security functionalities of various sorts, the implementation of a proper encryption/decryption module is essential. Section 2 presents a brief description of a generic embedded system as well as the Advanced Encryption Standard (AES) [8].

By their very nature, networked embedded (real-time) systems have significantly increased demands with respect to mobility, performance, resources, and being tamper proof. In terms of configuration flexibility and integrity enforcement, field-programmable gate arrays (FPGAs) seem to be an ideal implementation platform. FPGAs represent a piece of hardware that can be fully configured according to the designer's desires. Section 3 presents a brief overview about this technology.

From a developer's point of view, it is interesting to note that by using high-level hardware description languages, such as VHDL [16] or Verilog [17], the hardware can be configured (programmed) almost like an "ordinary" C program. Furthermore, existing hardware description files, also known as intellectual properties or IP cores, allow for the integration of high-level modules, such as network interfaces, memory controllers, co-processors (crypto, video, audio), I/O-controllers, and even entire processors, by just a few clicks. In other words, these tools make the development of custom hardware almost as easy as the development of simple C programs.

The flexibility of FPGAs as discussed above comes at the expense of runtime performance. State-of-the-art FPGAs allow a configured (soft-core) processor to be clocked at about 50 - 200 MHz, which might turn into a problem if aiming to realize real-time systems. Thus,

Section 4 discusses four options for the implementation of the AES encription/decryption functions in custom hardware and it also discusses how to seamlessly integrate this hardware into the entire system. Section 5 discusses how the integration of custom hardware affects the system's performance. It turns out that these enhancements improve the runtime by a factor of about 12 to four thousand. Finally, Section 6 concludes this paper with a brief discussion.

2. ENCRYPTION AT THE FIREWALL-ON-CHIP

The firewall-on-chip (FoC) [12] is an embedded security gateway for multi-level security [11], and is being developed in an ongoing research project at the University of Rostock. FoC aims at a complete high-security solution for networked embedded systems. The implementation of *interaction security* assumes that (1) the two interacting systems exchange messages that are encapsulated in (IP) packages and that (2) both sender and receiver have to pass all messages through their firewall-on-chip.

A firewall-on-chip enforces the security policy defined for its system, including authentication, access control, and cryptographic protection (encryption/decryption, integrity protection) of the packets sent and received. This includes the labeling and tagging of the messages according to the security classification of the information processed and its cryptographic separation (see Figure 1). At the wire level, this is achieved by means of a specifically developed protocol and message format, the Security Encapsulating IP Payload (SEIPP), which can, slightly simplistically, be seen as IPSEC [13] enhanced with classification tags.

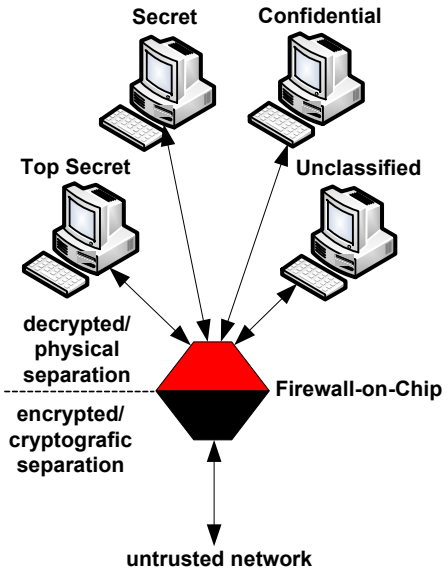

Figure 1. A standard multi-level security implementation by means of an embedded firewall-on-chip (FoC)

A proper integration concept must ensure that functions of the firewall-on-chip are non-bypassable, evaluable (for security certification), always be invoked (for each interaction), and tamper proof (the so called NEAT properties [14]). The firewall-on-chip is implemented on an FPGA at the connection point to the network so that no traffic can bypass its functions, it hence implements the access control, content filtering, and SEIPP packaging/unpackaging (including tagging/untagging, encryption/decryption, and message integrity code generation/evaluation) for all incoming and outgoing messages. The requirement for being tamper proof is the main motivation for the implementation of the security gateway as part of an FPGA. The internal structure of the firewall-on-chip as well as the usage of the (product-specific) FPGA-code-protection features ensure tamperproofness and makes it suitable for evaluation.

The bulk message encryption (and decryption) is a single security function that is computationally most expensive and thus best suited for the analysis and discussion of implementation alternatives on FPGAs. The block cipher algorithm used for information just classified as sensitive is the Advanced Encryption Standard (AES).

Advanced Encryption Standard: AES is a substitution-permutation network, which is a series of mathematical operations that use substitutions (also called S-Boxes) and permutations (P-Boxes) and ensure that each output bit depends on every input bit. AES is an iterated block cipher with a fixed block size of 128 and a variable key length of 128, 192, or 256 bits. The transformations per round operate on the intermediate results, called states. A state is a rectangular array of bytes. Since the block size is 128 bits, i.e., the rectangular array is of dimensions 4x4. Similarly, the round key, which is derived from the main key for each round, can be imagined as a rectangular array with four rows and four columns. The AES algorithm executes 10, 12, and 14 rounds for 128, 192, and 256 bits, respectively. During each round, the following operations are applied to the state:

1. SubBytes (SB): every byte in the state is replaced by another one, using the S-Box.
2. ShiftRow (SR): every row in the 4x4 array is shifted a certain amount to the left.
3. MixColumn (M): a linear transformation on the columns of the state.
4. AddRoundKey (A): each byte of the state is combined with the round key.

The description presented above is not meant to be exhaustive for the understanding of the AES algorithm, good descriptions can be found in most text books on cryptography or computer security [20]. However, this description should indicate the type and the number of

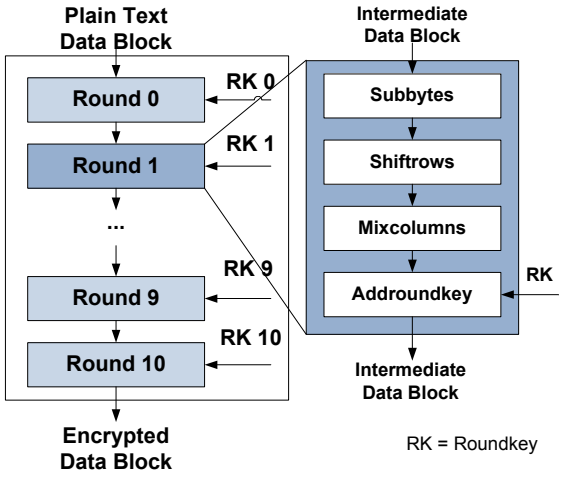

Figure 2. Advanced Encryption Standard (AES)

operations involved in the encryption of one 128 bit block. It should be noted that the computational complexity of the encryption and decryption operations are equivalent.

3. BACKGROUND: FIELD-PROGRAMMABLE GATE ARRAYS

This section provides a brief overview about field-programmable gate arrays (FPGAs) as far as necessary for the understanding of the present case study. This overview is target towards readers with only little or no FPGA experiences, and covers the internal architecture, the usual design flow, i.e., how an FPGA can be configured in order to achieve a certain hardware functionality, and the concept of soft-core processors.

3.1. THE TYPICAL FPGA ARCHITECTURE

In essence, an FPGA consists of a very large number of simple logic elements that map a small number of inputs onto an output port. From a functional point of view, every logic element employs a configuration register, which allows it to implement *any* logical function, such as AND, NAND, NOR, XOR, etc. From a technological point of view, this configuration register is an SRAM-based vector, which provides the element's output values for all input combinations. The logic element's implementation and the configuration vector is normally called look-up table (LUT) and LUT mask, respectively.

An FPGA provides not only the logic elements but also a significant number of configurable interconnections. These interconnections connect different logic elements and allow for the generation of more complex entities, such as state-machines, counters, adders, etc. These entities, in turn, can be combined to form complex modules or even entire systems, also called system-on-chip (SoC). In addition to the logic elements and its interconnections, most FPGAs also provide an infrastructure, which consists of one or more clock generators, I/O ports, and embedded memory cells.

State-of-the-art FPGAs host up to 150,000 logic elements. This number of logic elements is sufficient to realize complex modules, such as network adapters, floating-point units, and simple processors, combinations thereof as well as entire systems. For example, Altera's Nios II Development Kit Cyclone II Edition [4] contains 33,216 logic elements, charges about 995 USD, and is able to host about 4 - 8 processors, a network adapter, memory controllers for external memory, about 40 Kbytes on-chip memory, etc.

In contrast to other devices, such as read-only memories and application-specific integrated circuits (ASICs), an FPGA is always configured at startup time. To this end, an on-board configuration device reads a device configuration file and configures all its internal logic elements, routing resources and registers accordingly. Thus, an FPGA can provide a different functionality any time it is re-programmed with a different configuration file. Such a configuration file is several kilo bytes in size, and is the result of a design process described in the following subsection.

3.2. DESIGN FLOW

The design has the following two main purposes: (1) it has to functionally design the desired system and (2) it has to map the basic logical functions onto the FPGA's logic elements. By considering several hundreds of thousands of logic elements, this process is but easy. Therefore, the FPGA vendors also provide the designer with several tools such that the design can be done on various abstraction levels, which require different levels of expertise.

On the most abstract level, the designer is provided with graphical tools with which a system can be composed out of basic modules. This applies not only to choosing the components but also to their actual configuration, e.g., the number of a module's internal registers as well as their width. Afterwards, these tools do the transformation into an appropriate, low-level hardware description language. It might be interesting to note that the usage of these design tools is easy and straight forward.

On the lower abstraction level, the designer is using a hardware description language, such as VHDL or Verilog, which provides the designer with the most-possible design flexibility. Even though these languages are in design and usage quite similar to the C programming language, they require significant in-depth knowledge and thus require significant training.

The hardware designs produced by any of the methods mentioned above can be combined with already existing descriptions, which greatly improves the design efficiency. For example, an application-specific component can be easily combined with an existing soft-core processor. Given the low-level hardware description of all involved hardware modules, the development tools generate the appropriate LUT-masks (synthesis), assign every LUT-mask to a single logic element (placement), and determine the logic elements' interconnections (route). Finally, the result of the process is stored into the configuration file that in turn can be transferred to the FPGA.

3.3. THE NIOS II SOFT-CORE PROCESSOR

A simple soft-core processor, i.e., a regular CPU without instruction pipelining, multi-level interrupt processing, floating-point co-processors, etc., requires about 700 – 1,800 logic elements for its implementation. It is thus easy for an FPGA to host one or even a small number of them. Because of the prevailing importance of processors in most hardware designs, FPGA vendors as well as others provide several soft-core processors. These soft-core processors either execute standard operation codes, or provide their own cross-compilers. Some well-known examples are Nios and Nios II CPUs from ALTERA [5], LEON CPUs from Gaisler Research [9], and the Microblaze CPU from XILINX [10].

The case study presented in this paper resorts to the Nios II processor that is shipped with the Altera Cyclone, Stratix and Hardcopy device families. This 32-bit processor has up to 64 Kbytes cache, can be clocked at up to 200 MHz, includes hardware interrupts, and hardware multiplication units.

A key property of soft-core processors is that they are given in a hardware description language. The designer is thus able to easily fine tune the processor's architecture to the application at hand. In addition, the designer is also able to freely combine the processor with other hardware components on the very same chip. These extra components can either be connected via a system bus, or can be tightly integrated into the actual processor. On the one hand, the various design options differ with respect to the achievable performance figures, but have on the other hand different requirements with respect to both design efforts and hardware consumption. The exploration of these options in the context of tamper-proof hardware is subject to the case study presented in the following sections.

Summary: Field-programmable gate arrays hardware devices that can be easily (re-) configured without inducing significant development costs, and have their advantages in providing special-purpose and parallel processing solutions as well as in offering hardware protection mechanisms. Thus, FPGAs might be a good complement for standard, PC-based (software) systems.

4. HARDWARE-SOFTWARE CO-DESIGN

When developing an FPGA-based embedded system, hardware and software are concurrently designed to ensure compatibility. The hardware imposes several software relevant constraints, such as memory space, number of interrupts, and I/O-port definitions. To keep track of these constraints, especially after hardware modifications, design and programming is usually supported by integrated development environments. These tools allow, for example, for the graphical configuration of a soft-core processor, performing all FPGA-related work (synthesis, placement, FPGA-configuration), updating software header files, and compilation and download of the software for its execution within the FPGA-internal soft-core processor. Figure 3 gives a schematic overview of these single steps.

In the present case study, the easiest way would be to implement a soft-core processor, which among other things, executes the AES algorithm. Section 2 has already indicated that the AES algorithm is of a high computational complexity. Since every single (IPSEC) network packet has to be processed by this module, a relatively low throughput is most likely. It would thus be quite better to implement certain AES core functionalities directly in hardware. The hardware implementation of all security-sensitive parts of the AES algorithm would furthermore increase the level of security of the overall system. For the endeavor, the designer has the following two main options:

Automated hardware generation: Altera offers an automated software-to-hardware conversion tool, called C2H compiler. This compiler is capable of generating hardware modules based on a given C source code. The usage of this compiler is quite simple. After choosing a C function, the C2H compiler generates a functionally equivalent hardware module and integrates it into the soft-core processor. This change in the system's hardware requires a re-synthesis of the FPGA-system; the compiler executes all relevant updates, which includes the configuration files, the software headers, and so forth. Finally, the compiler replaces the body of the scheduled C function with a system call to the newly integrated hardware module. Thus, from a software perspective no further changes are necessary. All the other parts of the software can make use of the "old" function as usual. The utilization of the hardware module is entirely "hidden" within the function.

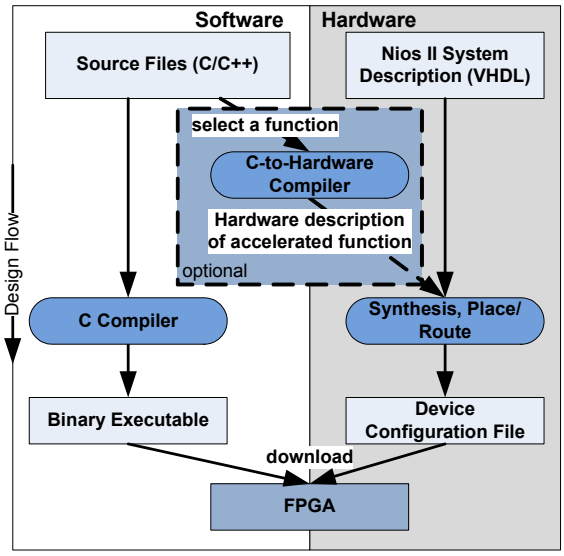

Figure 3. Hardware-Software co-design with C-to-Hardware Compiler

For its hardware conversion process, the C2H compiler applies very strict conversion rules [3]. For example, the compiler allows for the usage of only a subset of the C standard data types, such as integer, char, etc., but not float or double. This in turn imposes limitations on the achievable performance improvements. Thus, although the designer does not need any hardware expertise for this task, a short look at the conversion rules of the C2H compiler and additional restructuring of the source code may improve the results.

In addition, the automated hardware generation for rather large C functions also imposes high demands on the C2H compiler: The compiler has to adhere to both data dependencies and timing constraints of the original C variables. Thus, complex C functions lead to complex hardware, i.e., hardware that requires many resources on the FPGA. As a consequence, the ratio of the performance improvement and the amount of utilized hardware resources may benefit from an incremental conversation of the selected C functions. This can be achieved by simply splitting up large code fragments into a set of simpler sub-functions, and converting those according to their influence on the performance of the software.

Hand Coding: Experienced designers can generate arbitrary functionalities at their own disposal. In so doing, the designer can modify existing hardware descriptions (i.e., modules) or can develop new ones from scratch. This method uses a hardware description language, like VHDL or Verilog, to describe the logical structure of each component. A detailed knowledge about the system architecture, bus functions, and interfaces is required. The integration of the manually designed components into the system also requires slight modifications within the program code. The former function is replaced by an appropriate hardware call.

This hand coding approach offers further options to enhance the performance of the system. Modules generated by the C2H compiler are always connected to the internal system bus (see Fig. 5). But the experienced designer can choose from three options of integrating new hardware modules within the system: (1) The component can be attached to the system bus. (2) The component can reside outside the soft-core processor. This approach allows, for example, for two different clock domains: one for the soft-core processor and another one for the new component. The software calls the hardware via a standard bus interface. (3) The component is tightly integrated into the soft-core's central processing unit, i.e., as a parallel data path to the arithmetical logical unit. In so doing, the data signals from the CPU to the hardware module are as short as possible, making communication as fast as possible. This is referred to as *custom instruction* [7]. The development environment generates a unique assembler instruction for each custom instruction during the FPGA-synthesis.

Where as the C2H compiler applies a strict rule set during the conversation, an experienced designer may freely change the hardware module's structure. This refers to hardware issues, such as pipelining and parallelism as well as changing the entire structure of the algorithm to achieve performance enhancement.

Another main benefit of hardware-software co-design is testability: Due to the existence of both, a software *and* a hardware solution, results can be easily compared. The ability of keeping the development process as short as possible is rather important, especially when taking into account that test and verification of hardware components consumes about 70% of the traditional hardware design cycles [19].

5. IMPLEMENTATION OF AES AND ITS RESULTS

This section presents the architectures of five different hardware-software co-designs. The architectures differ in the amount of how much functionality has been realized in hardware as well as the resulting processing performance. The performance results have been achived with Altera's low-cost Cyclone-II FPGA [4], which offers 33,216 logic elements.

Nios II soft-core processor: The first, most straight forward design consists of a standard Nios II soft-core processor, which utilizes the 16MB DDR RAM as a conventional data and instruction memory. This design is presented in Figure 4a, and serves as a reference point for all experiments presented in this paper.

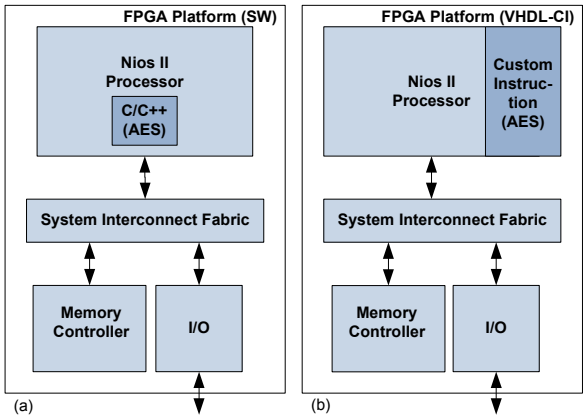

Figure 4. (a) FPGA platform with Nios II processor and (b) FPGA platform with custom instruction

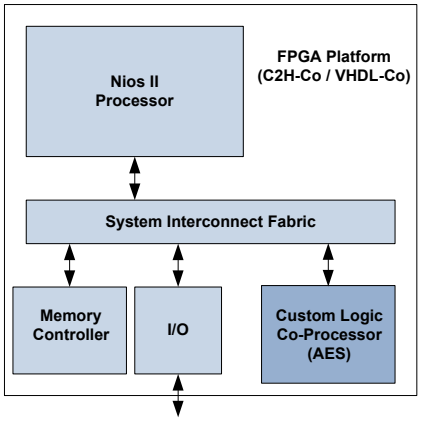

Figure 5. FPGA platform with a custom logic co-processor

In this design, the soft-core processor executes a monolythic software imlementation of entire AES algorithm [18]. The practical experiments have shown that this system can only be clocked at about 85 MHz. Even though this soft-core processors employs an instruction cache of 4 kilo bytes as well as embedded multipliers, the achievable throuput was as low as only 9 Kbits/s.

Automatically generated co-processor (C2H-Co): The first hardware alternative was developed by applying Altera's C2H compiler to the `ciper()` function of the AES algorithm. The result is a separate encryption module, which communicates with the soft-core processor by means of an existing on-chip communication bus. This communication bus is called Avalon. This design is presented in Figure 5, and denoted as C2H-Co for short.

This by the C2H compiler automatically generated co-processor module accelerates the execution of the AES algorithm by a factor of about 37, which is quite good. However, this module consumes about twice as much resources as the rest of the system, which is a quite significant portion of the available hardware. But as has already been mentioned in Section 4, the C2H compiler can be used by almost any designer.

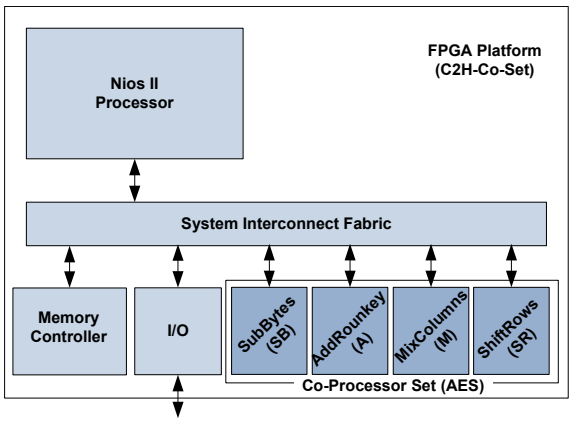

Figure 6. FPGA platform with a custom logic co-processor set

Automatically generated set of sub-modules (C2H-Co-Set): The second design alternative applies the C2H compiler to the AES algorithm's main functions, which are `addroundkey()`, `subbytes()`, `shiftrows()`, and `mixcolumns()`. This design is shown in Figure 6, and denoted as C2H-Co-Set for short. In the experiments, this design yields a speedup of only 12, which is approximately a third of the C2H-Co's. This reduced speedup is due to the soft-core processor, which is left with an additional communication overhead. The four co-processors requiere as much logic elements as the rest of the system, i.e., compared to the C2H-Co implementation only half as much additional logic elements are used.

Despite the performance improvement, the following quite subtile point might be of interest: half of the achieved speedup is due to the `subbytes()`-function. That means that even if all other functions remain be implemented in software, 1600 logic elements, which is only one forth of the C2H-Co-Set, are able to accelerate the execution of the cipher algorithm six times. Considering the small number of required logic elements, the transformation of this function leads to a very good ratio of speedup and consumed resources, also called efficiency factor.

Figure 8 shows that, the more functions are implemented in hardware, the speedup increases while the efficiency decreases. The reason is that the hardware implementation of the `shiftrows()`-function, for example, comes only with an additional speedup less than one but with 1500 logic elements.

Hand-coded Co-Processor (VHDL-Co): In the third hardware design, a designer has realized the AES co-processor using a hardware description language, such as VHDL. In so doing, the designer has done both tailoring the implementation towards the given

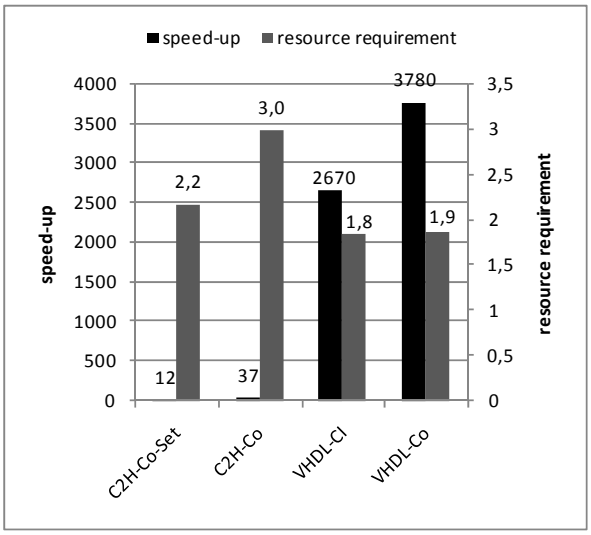

Figure 7. Speed-up and resource requirement of the hardware implementations compared to the soft-core implementation

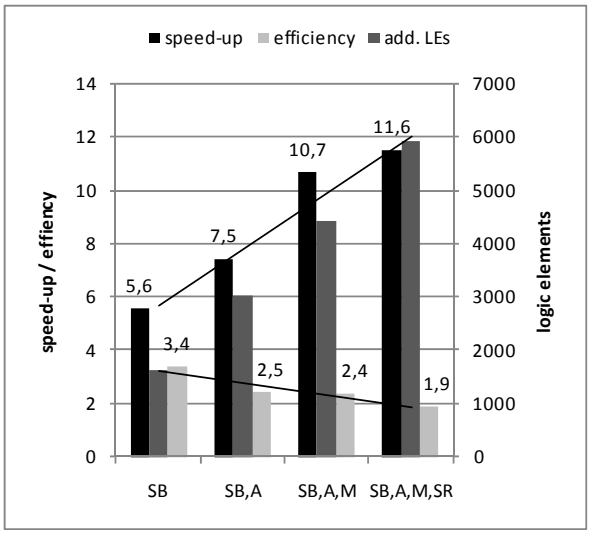

Figure 8. Speed-up and efficiency of different C2H-Co-Set implementations

hardware architecture and exploiting some properties of the AES algorithm. This design is equivalent to the one already shown in Figure 5, and is denoted as VHDL-Co for short.

This design accelerates the execution of the AES algorithm by a factor of almost four thousand, and requires only 4400 additional logic elements. In other words, this co-processor is about a hundret times faster than the automatically generated co-processor and is only half of its size. However, in order to be successful, this option requires a very experienced hardware designer.

Hand coded processor extension (VHDL-CI): Finally, the forth design option directly integrates the hand-coded `cipher()`-function into the soft-core processor. This tightly integrated encryption/decryption module can be used by means of a *custom instruction*. This architecture is presented in Figure 4b, and called VHDL-CI for short. This forth design alternative yielded similar performace improvements as the VHDL-Co design did.

Table 1. Results

	SW	C2H-Co-Set	C2H-Co	VHDL-CI	VHDL-Co
throughput at 85 MHz (MBits/s)	0.009	0.108	0.344	24.802	35.112
speed-up	1	12	37	2670	3780
required logic elements	5032	10980	15063	9282	9418
resource requirement	1.0	2.2	3.0	1.8	1.9

Summary and Discussion: The results are summarized in Table 1. This table compares the five developed architectures with respect to the achieved speedup, the number of required logic elements, as well as their ratios with respect to the software solution, i.e., the soft-core architecture shown in Figure 4a.

The presented data, depicted in figure 7, shows that hand coded AES modules yield a speedup, which is about 100 times better than the ones achieved by using the C2H compiler. The data also shows that hand coded solutions require about 50 % fewer additional logic elements. From a management point of view, these results are interesting in that the C2H compiler can be successfully used by almost *any* designer, where as the further improvements require substantial expert knowledge. However, if aiming at the design of networked embedded *real-time* systems, this additional training in a hardware description language, such as VHDL or Verilog, might be worth it. But in any case, all four hardware-based implementations significantly improve the system's security, since the AES algorithm cannot be changed after deployment, at least not with reasonable efforts.

It might be mentioned that all five implementations can be further improved by applying some additional optimizations as proposed by the vendors' guidelines. In addition, all hardware implementations can be further improved by using design concepts, such as parallel processing and code pipelining. However, since this paper focuses on using hardware-software co-design techniques, these design optimizations are beyond the scope of this paper.

6. CONCLUSION

This paper has discussed several options of how field-programmable gate arrays can improve the design of tamper-proof networked embedded systems. To this end, this paper has focused on the implementation of a security essential, the Advanced Encryption Standard (AES) algorithm.

From a security point of view, the main advantage of using field-programmable gate arrays is that they cannot be altered (modified) neither by internal nor by external attacks. In addition, the usage of field-programmable gate arrays has also advantages from a developer's point of view. On the module level, the design is almost as easy as developing a simple C program. Furthermore, designers have access to some powerful tools that provide excellent design support.

The present case study has also discussed several options with which the designer can implement certain functionalities directly in hardware. The case study on the implementation of the AES algorithm has shown that the C2H compiler, which is very easy to use, yielded a speedup of about 12 to 37 but increased the design size by a factor of about 2 to 3. By contrast, the direct usage of VHDL, which requires expert knowledge, was able to accelerate the processing by a factor of about four thousand, which came at a cost of a doubled design size.

The results of the present case study suggest that the C2H compiler provides good services if the following conditions are met: (1) The goal is a low performance improvement by a factor of 10 to 50, (2) no VHDL expert knowledge is available, and (3) an immediate implementation is necessary, i.e., small design and test time, and (4) there are enough resources available on the FPGA.

The implementation of the function by means of direct VHDL is useful under the circumstances listed below: (1) A significant performance improvement is required, (2) VHDL expert knowledge is available, (3) there is enough design and test time available, and (4) FPGA resources are plenty.

Acknowledgements

The authors gratefully thank Marcus Wittig for his good cooperation as well as numerous fruitful discussions. This research was supported in part by the Federal Ministry of Economy and Technology, grant number KF0451702SS7.

REFERENCES

[1] Altera Corp., *Nios II Software Developer's Handbook*, Altera Document NII52001-7.2.0, Altera Corp., San Jose, CA, 2007.

[2] Altera Corp., *Nios II C2H Compiler User Guide*, Altera Document UG-N2C2HCMPLR-1.3, Altera Corp., San Jose, CA, 2007.

[3] Altera Corp., *Optimizing Nios II C2H Compiler Results*, Altera Document AN-420-1.0, Altera Corp., San Jose, CA, 2006.

[4] Altera Corp., *Nios Development Board Cyclone II Edition Reference Manual*, Altera Document MNL-N051805-1.3, Altera Corp., San Jose, CA, 2007.

[5] Altera Corp., *Nios II Processor Reference Handbook*, Altera Document NII5V1-7.2, Altera Corp., San Jose, CA, 2007.

[6] Altera Corp., *Quartus II Version 7.2 Handbook Volume 4: SOPC Builder*, Altera Document QII5V4-7.2, Altera Corp., San Jose, CA, 2007.

[7] Altera Corp., *Nios II Custom Instruction User Guide*, Altera Document UG-N2CSTNST-1.4, Altera Corp., San Jose, CA, 2007.

[8] United States Federal Information Processing Standards Publication 197, *Advanced Encryption Standard (AES)*, 2001.

[9] Gaisler Resarch, *LEON2 Processor User's Manual -XST Edition*, Gaisler Research AB, Goteborg, Sweden, 2005

[10] XILINX Inc., *MicroBlaze Processor Reference Guide, XILINX Document UG081 (v5.0)*, XILINX Inc., San Jose, CA, 2005

[11] R. Smith, "Introduction to Multilevel Security", In *Handbook of Information Security*, Vol. 3, H. Bidgoli, ed., 2005.

[12] Web Site Firewall-on-Chip project: http://www.imd.uni-rostock.de/index.php?id=262

[13] S. Kent, K. Seo, "Security Architecture for the Internet Protocol", *Internet RFC 4301*, 2005.

[14] J. Alves-Foss, P.W. Oman, C. Taylor, C.W. Harrison, "The MILS architecture for high-assurance embedded systems", In *International Journal of Embedded Systems*, Vol. 2, No.3/4, pp. 239 – 247, 2006.

[15] B. Snow, "Four Ways to Improve Security", *IEEE Security and Privacy*, Vol. 3, No. 3, pp. 65-67, 2005.

[16] VHDL Analysis and Standardization Group, *IEEE Standard VHDL Language Reference Manual*, IEEE Standard 1076-2002, 2002.

[17] IEEE Computer Society, *IEEE Standard for Verilog Hardware Description Language*, IEEE Standard 1364 - 2005, 2005.

[18] Advanced Encryption Standard (AES) Implementation in C/C++, www.hoozi.com/Articles/AESEncryption.htm

[19] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*, Second Edition, Kluwer Academic Publishers, 2nd edition, 2003

[20] D. R. Stinson, *Cryptography : Theory and Practice*, Chapman & Hall / CRC, 2nd edition, 2002