ASIP Architecture Exploration for efficient IPSec Encryption: A Case Study

Hanno Scharwaechter, David Kammler, Andreas Wieferink, Manuel Hohenauer, Kingshuk Karuri, Jianjiang Ceng, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr

Integrated Signal Processing Systems,
Aachen University of Technology,
Aachen, Germany
scharwaechter@iss.rwth-aachen.de

Abstract. *Application Specific Instruction Processors* (ASIPs) are increasingly becoming popular in the world of customized, application-driven *System-on-Chip* (SoC) designs. Efficient ASIP design requires an iterative architecture exploration loop - gradual refinement of processor architecture starting from an initial template. To accomplish this task, design automation tools are used to detect bottlenecks in embedded applications, to implement application-specific instructions and to automatically generate the required software tools (such as instruction set simulator, C-compiler, assembler, profiler etc.) as well as to synthesize the hardware. This paper describes an architecture exploration loop for an ASIP coprocessor which implements common encryption functionality used in symmetric block cipher algorithms for *IPsec*. The coprocessor is accessed via shared memory and as a consequence, our approach is easily adaptable to arbitrary processor architectures. In the case study, we used Blowfish as encryption algorithm and a MIPS architecture as main processor.

1 Introduction

The strong growth of internet usage during the past years and the resulting packet traffic have put tight constraints on both protocol and hardware development. On the one hand, there is the demand for high packet throughput and on the other hand, protocols have to meet the continously changing traffic requirements like Quality-Of-Service, Differentiated Services, etc. Furthermore, the increasing number of mobile devices with wireless internet access like laptops, PDAs and mobile phones as well as *Virtual Private Networks* (VPNs) has made security one of the most important features of today's networks. *IPsec* is probably the most transparent way to provide security to the internet traffic. In order to achieve the security objectives, *IPsec* provides dedicated services at the *IP* layer that enable a system to select security protocols, determine the algorithm to use, and put in place any cryptographic keys required. This set of services provides access control, connectionless integrity, data origin authentication, rejection of replayed packets (a form of partial sequence integrity), confidentiality

(encryption) and limited traffic flow confidentiality. Because these services are implemented at the *IP* layer, they can be used by any higher layer protocol, e.g. TCP, UDP, VPN etc.

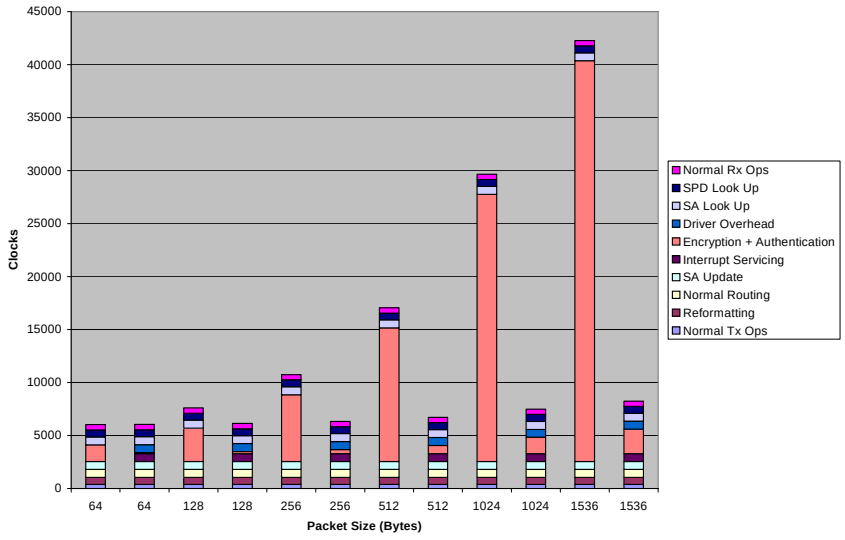

Fig. 1. Break-up of tasks in typical VPN traffic.

Integrating security warranties into the *IP* stack inevitably influences overall *IP* processing performance. Fig. 1 shows break-ups of tasks in implementations of VPN and their execution time related to the packet size. The columns alternately represent implementations of VPN in software and hardware, starting with a software implementation with an incoming packet size of 64 bytes. Since *data encryption* is the most computation intensive task in *IPsec*, it becomes one of the most promising candidates to increase overall packet processing performance. But encryption algorithms are an ever-changing area of computer science. Regularly they are cracked or replaced by newer ones. For example, currently the *Data Encryption Standard* (DES) is replaced by the *Advanced Encryption Standard* (AES). Implementing such algorithms completely in hardware (i.e as a separate *Application Specific Integrated Circuit* (ASIC)) is not feasible due to a lack of reuse opportunities. A good compromise between flexibility and efficiency are *Network Processing Units* (NPUs) which constitute a subclass of ASIPs. They offer highly optimized instruction sets tailored to specific network application domains (which in our case is encryption). Overall performance can be further enhanced by including specialized coprocessors to perform tasks like table lookup, checksum computation, etc. and by expanding the data path to support necessary packet modifications.

In order to design efficient NPUs like any other ASIPs, *design space exploration* (fig. 2) at the processor architecture level needs to be performed [7],[8]. Architecture exploration usually starts with an initial architectural prototype. The pure software implementation of the intended application is run and profiled on this prototype to determine probable performance bottlenecks. Based on the profiling results, the designer refines the basic architecture step by step (e.g. by adding custom instructions or fine-tuning the instruction pipeline) until it is optimally tailored towards the intended range of applications.

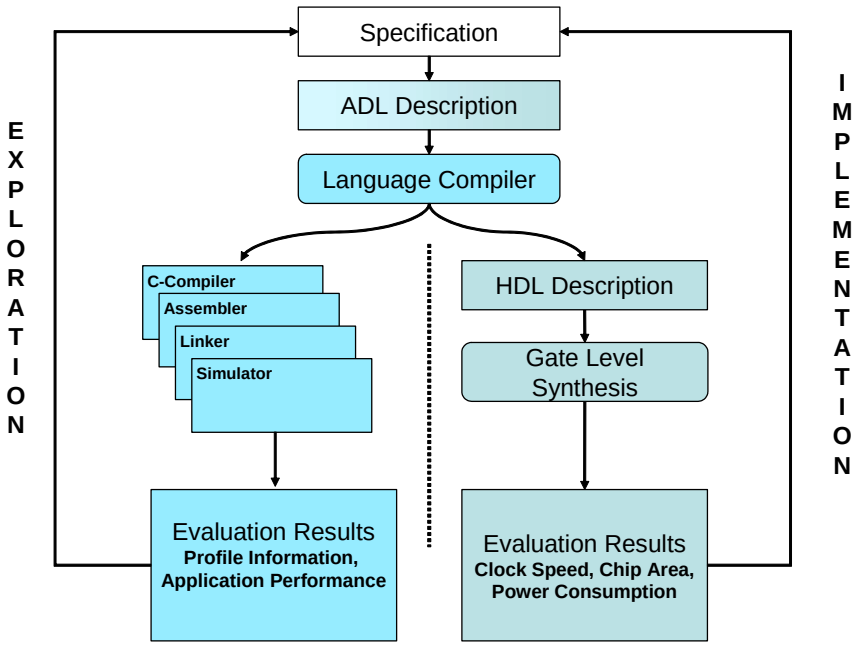

Fig. 2. Tool based processor architecture exploration loop

This iterative exploration approach demands for very flexible *retargetable* software development tools (C-compiler for main processor, assembler, cosimulator/debugger etc.) that can be quickly adapted to varying target processor-coprocessor configurations as well as a methodology for efficient *Multiprocessor-SoC* exploration on system level. Retargetable tools permit to explore many alternative design points in the exploration space within short time, i.e. without the need of tedious complete tool re-design. Such development tools are usually driven by a processor model given in a dedicated specification language.

Although this approach is the basic reason for the success of *Architecture Description Languages* (ADLs), the link to the physical parameters such as chip area or clock speed gets lost. The necessity of combining the high level abstraction and physical parameter evaluation in a single exploration is obvious.

In this paper we present an architecture exploration case study for a coprocessor, supporting symmetric block cipher functionality using an ADL tool suite. Our main intention is to show the usage of the tools and their interaction. The remainder of this paper is organized as follows: In section 2 we give a short introduction to our tool suite that we used within this exploration followed by a discussion of the related work in section 3. The main body of the paper consists of the illustration of our target application with main focus on the encryption algorithm in section 4, followed by the detailed presentation of the successive refinement flow for the joint processor/coprocessor optimizations in section 5 as well as the obtained results. Section 6 concludes the paper.

2 System Overview

The architecture exploration framework presented in this paper builds on the *LISATek Processor Designer*, a tool platform for embedded processor design available from CoWare Inc. [4], an earlier version of which has been described in detail in [8]. The *LISATek* tool-suite revolves around the LISA 2.0 ADL. Amongst others, it allows for automatically generating efficient ASIP software development tools like instruction set simulator [12], debugger, profiler, assembler, and linker, and it provides capabilities for VHDL and Verilog generation for hardware synthesis [15]. A retargetable C-compiler [14] [1] is seamlessly integrated into this tool chain and uses the same single "golden reference" LISA model to drive retargeting. A methodology for system level processor/communication coexploration for multi-processor systems [21] is integrated into the LISA tool chain, too. We believe that such an integrated ADL-driven approach to ASIP design is most efficient, since it avoids model inconsistencies and the need to use various special-purpose description languages.

3 Related Work

The approaches that come closest to ours are Expression [7], ASIP Meister [13], and CHESS [9]. Similar to our approach with the LISA language, Expression uses a dedicated, unified processor ADL with applications beyond compiler retargeting (e.g. simulator generation). *Hardware Description Language* (HDL) generation from the Expression language is presented in [7]. The HDL generation is based on a functional abstraction and thus allows to generate the complete architecture.

Like our approach, ASIP Meister builds on the CoSy compiler platform [1]. However, it has no uniform ADL (i.e. target machine modeling is completely based on GUI entry) and the range of target processors is restricted due to a predefined processor component library. That is why the HDL generation of ASIP Meister is able to fulfill tight constraints regarding synthesis results, but

[1] Based on CoSy compiler development system from ACE. [1]

sacrificing flexibility.

CHESS uses the nML ADL [6] for processor modeling and compiler retargeting. Unfortunately, only few details about retargeting CHESS have been published. Like LISA, nML is a hierarchical mixed structural/behavioral ADL that (besides capturing other machine features) annotates each instruction with a *behavior description*. While LISA permits arbitrary C code for behavior descriptions, such descriptions in nML are restricted to a predefined set of operators which probably limits the flexibility of CHESS. Build on nML, an HDL generator GO from Target Compiler Technologies [19] exists which results are unfortunately not publicly available. Furthermore, the project Sim-HS [20] produces synthesizable Verilog models from Sim-nML models. Here, non-pipelined architectures are generated and the base structure of the generated hardware is fixed. Additionally, to our knowledge none of the here mentioned frameworks provides support for retargetable MP-SoC integration at the system level, which was, due to our processor-coprocessor design, a major drawback.

There are several existing architectures for cryptographic coprocessors targeted towards embedded systems. Some general architectures are presented in [3], [11], [2], [10]. Chodowiec [3] and his group show that advanced architectural techniques can be used to improve performance for block cipher algorithms; they implement pipelines and loop-unrolling in an architecture based on a *Field Programmable Gate Array* (FPGA). Similar approaches are taken in [11] and [2] with regards to the *International Data Encryption Algorithm* (IDEA). They compare the differences in performance of serial and parallel implementations of IDEA using a Xilinx Virtex platform; power consumption is not considered. In [10], the authors explore a methodology for hardware-software partitioning between ASICs, *Digital Signal Processors* (DSPs) and FPGAs to optimize performance for customized encryptions units. The author of [10] also points out that since there are limited resources available to mobile communication devices, proper balance of performance and flexibility is important. He presents design choices associated with these factors in his FPGA-based implementation of IDEA.

4 Target Application

The *Internet Protocol* (IP) is designed for use in interconnected systems of *packet-switched* computer communication networks. It provides facilities to transmit blocks from sources to destinations which are identified by fixed length addresses. The protocol is specifically limited in scope to provide the functions necessary to deliver a datagram from source to destination, and there are no mechanisms for other services commonly found in host-to-host protocols.
Because of the need for an upgrade anyway, it was logical that the new version of the internet protocol - *IPv6* - should contain a native security system which would allow the users to communicate securely. At the same time, it must be

realized that because the internet is a vast and complex network, the transition to the new version of the protocol will not be immediate. Hence the security implementation should be such that it would be compatible, and adaptable to *IPv4*.

IPsec focuses on the security that can be provided by the IP-layer of the network. It does not concern itself with application level security such as *Pretty Good Privacy* (PGP), for instance.

Security requirements can be divided into two distinct parts:

- Authentication & Integrity and
- Confidentiality.

These are independent of each other and can be used separately or together according to user requirements. The encryption and authentication algorithms used for *IPsec* are the heart of the system. They are directly responsible for the strength of the security the system can provide. *IPsec* generally claims for block cipher algorithms which support *Cipher Block Chaining* (CBC) mode [16]. Roughly spoken, this means that the encryption of a certain block of data is affected by the encryption of preceding blocks.

Our main application is the publicly available network stack implementation developed by Microsoft Research [5] known as *MSR IPv6*. To enhance *IPv6* performance, we identified and extracted the common path through this protocol including *IPsec* encryption. Based on this, we wrote an *IPv6* testbench. For the encryption we selected the Blowfish encryption algorithm.

Blowfish is a symmetric block cipher with 64-bit block size and variable length keys (up to 448 bits) designed by Bruce Schneier [16]. It has gained a wide acceptance in a number of applications. No attacks are known against it. This cipher was designed specifically for 32-bit machines and is significantly faster than DES. One of the proposed candidates for the AES called Twofish [17] is based on Blowfish. As most block cipher algorithms, Blowfish is a so called Feistel-Network which takes a block of size n, divides it in two halves of size $n/2$ and executes an iterative block cipher of the form

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_i \oplus F(R_{i-1}, K_i) \end{aligned}$$

where K_i is a partial key of ith round,
L, R are the right and left halves, respectively, of size $n/2$, and
F an arbitrary round function.

Feistel-Networks guarantee reversibility of the encryption function. Since L_i is *xor*-ed with the output of f, the following holds true:

$$L_{i-1} \oplus F(R_{i-1}, K_i) \oplus F(R_{i-1}, K_i) = L_{i-1}$$

The same concepts can be found in DES, Twofish, etc. Blowfish supports all known encryption modes like CBC, ECB OFB64, etc. and is therefore a good

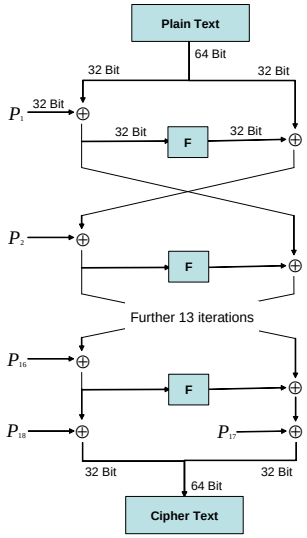

Fig. 3. Blowfish

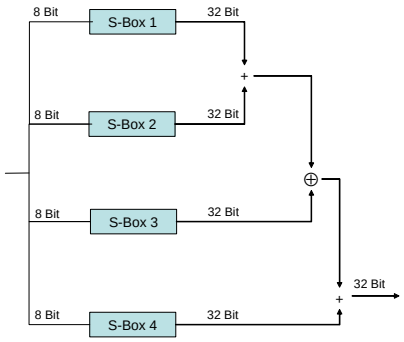

Fig. 4. Function F

candidate for *IPsec* encryption. Two main parts constitute the Blowfish encryption algorithm (fig. 3): *key expansion* and *data encryption*. The key expansion divides a given key into different 32-bit subkeys. The main key is 4168 bits wide and has to be generated in advance.

On the lowest level, the algorithm contains just the very basic encryption techniques *confusion* and *diffusion*[16]. *Confusion* masks relationships between plain and cipher text by substituting blocks of plain text with blocks of cipher text. Diffusion distributes redundancies of plain text over the cipher text by permuting blocks of cipher text. *Confusion* and *diffusion* depend strongly on the set of subkeys. 18 subkeys constitute a permutation array (P-array), denoted as

$$P_1, P_2, \ldots, P_{18}$$

for *confusion*. Four substitution arrays (S-Boxes) - each of 256 entries - denoted as

$$S_{1,0}, S_{1,1}, \ldots, S_{1,255}$$
$$S_{2,0}, S_{2,1}, \ldots, S_{2,255}$$
$$S_{3,0}, S_{3,1}, \ldots, S_{3,255}$$
$$S_{4,0}, S_{4,1}, \ldots, S_{4,255}$$

control diffusion.

The *data encryption* is basically a very simple function (fig. 4) which is executed 16 times. Each round is made of a key dependent permutation, as well as a key and data dependent substitution which constitute the very basic encryption techniques. The used operations are either additions or *xor*-connections and four memory accesses per round. The exact encryption procedure works as follows:

Divide x into two 32-bit halves x_L and x_R
For $i = 1$ to 16:
    $x_L = x_R \oplus P_i$
    $x_R = F(x_L) \oplus x_R$
    Exchange x_L and x_R
Exchange x_R and x_L (reverts first exchange)
$x_R = x_R \oplus P_{17}$
$x_L = x_L \oplus P_{18}$
Concatenate x_L and x_R

The function F looks like the following:

Divide x_L into four 8-bit-quarter a,b,c and d.
$F(X_L) = ((S_{1,a} + S_{2,b} \bmod 2^{32}) \oplus S_{3,c}) + S_{4,d} \bmod 2^{32}$,

where $S_{i,j}$ designates index j of S-Box i for $i \in \{1 \dots 4\}$ and $j \in \{0 \dots 255\}$. Decryption works exactly in the same way, just with the difference that $P_1, P_2, \dots, P_{18}$ are used in reversed order.

5 Exploration Methodology

The key functionality of the LISA processor design platform is its support for *architecture exploration*: In the phase of tailoring an architecture to an application domain LISA permits a refinement from profiled application kernel functionality to cycle accurate abstraction of a processor model. This process is usually an iterative one that is repeated until a best fit between selected architecture and target application is obtained. Every change to the architecture specification requires a complete new set of software development tools. Such changes, if carried out manually, will result in a long, tedious and extremely error-prone exploration process. The automatic tool generation mechanism of LISA enables the designer to speed-up this process considerably. The design methodology is composed of mainly three different phases: *application profiling*, *architecture exploration* and *architecture implementation phase*.

5.1 Application Profiling

The *application profiling phase* covers tasks to identify and select algorithm kernels which are candidates for hardware acceleration. Such kernels constitute the

performance critical path of the target application and can be easily identified by instrumenting the application code in order to generate high-level language execution statistics by simulating the functional prototype.
For this purpose, we generated a C-compiler for a MIPS32 4K architecture by applying the LISA Compiler-Generator on the related LISA model, implemented our target application and profiled it with the LISA Profiler to obtain a general idea about bottlenecks and of possible hardware accelerations. The outcome was a pure functional specification of the instructions to be implemented. As expected, it turned out that most of the execution time is spent in the encryption algorithm. More detailed, 80% of the computations are spent in the above mentioned F function according to its iterative execution.

5.2 Architecture Exploration

During the *architecture exploration phase* (fig. 2 in section 1), software development tools (i.e. C-compiler, assembler, linker, and cycle-accurate simulator) are required to profile and benchmark the target application on different architectural alternatives. Using the C-compiler, the software and architecture designers can study the application's performance requirements immediately after the algorithm designer has finished his work.

To support Blowfish encryption most efficiently, but without complete loss of flexibility to develop other symmetric *IPsec* encryption algorithms, we decided to implement an encryption specific instruction set processor. Furthermore, we wanted to provide a maximum amount of flexibility concerning hardware constraints, which led to a coprocessor design, that is accessed by its host processor via shared memory and provides special purpose encryption instructions. This implies that the coprocessor has to run at least with the same clock speed as the main processor, because otherwise the original implementation of the IP stack would be slowed down by the coprocessor.

In fig. 5, the resulting dual-processor platform is depicted. For both processors, a processor simulator automatically generated from a LISA model is applied. The processor models contain two bus instances, one for the program memory requests and one for the data accesses. For high simulation performance, the memory modules local to one processor are modeled inside the respective LISA model, e.g. the kernel segment ROM (kseg_rom) and the user segment ROM (useg_rom) of the MIPS32 main processor. Only the memory requests to the shared memory are directed to the SystemC world outside the LISA simulators.

The platform communication is modeled efficiently using the *Transaction Level Modeling* (TLM) paradigm [18] supported by SystemC 2.0. A LISA port translates the LISA memory API requests of the processors to the respective TLM requests for the abstract SystemC bus model. The bus model performs the bus arbitration and forwards the processor requests to the shared memory. This memory is used to communicate between the processors and thus exchange parameters as well as results of the encryption procedures implemented on the

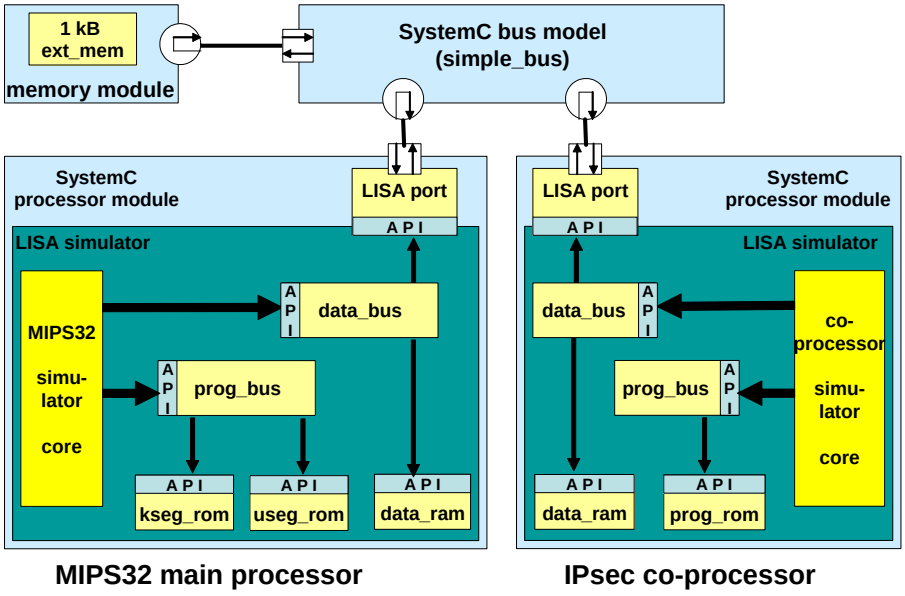

Fig. 5. Simulation setup

coprocessor and the original *IPv6* protocol stack implemented on the MIPS main processor. To access the coprocessor on C-code level, we extended the generated MIPS C-compiler by dedicated intrinsic functions, one for the encryption and the decryption procedure. These intrinsic functions have the same look and feel as the original functions of blowfish, but internally they push their parameters onto the shared memory block, poll a certain memory address, that indicates the end of the computation and pop the results from the shared memory block back to local memory for further processing on the MIPS. On the coprocessor side, the external memory is also polled for parameters. If now certain paramters are found in the memory, the relevant computation is performed and the results are written back to the external memory.

Since F is iteratively executed within the encryption, the decryption and also in the key generation procedures of Blowfish, interleaved parallel execution of F in a dedicated pipeline is not possible, because results from iteration i are used as input in iteration $i+1$. Another option was to develop instructions that cover partially behavior of F presented in fig. 4 and which can be executed in one cycle related to the MIPS cycle length. Therefore, our first design decision was to start from a RISC architecture including a 4-stage pipeline with fetch, decode, execution and writeback stage as in the initial LISA-model template. In the further discussed architecture co-exploration loops, the coprocessor core has been successively refined in order to reach a certain degree of efficiency. In the

following, the required architecture co-exploration loops are discussed in detail.

Implementing the instructions, we started with a first educated guess and divided the function F (fig. 4) into four independent parts, each of which can be executed in one execution stage:

$$
\begin{aligned}
u &= S_{1,a} \\
v &= S_{2,b} + u \\
w &= S_{3,c} \oplus v \\
x &= S_{4,d} + x.
\end{aligned}
$$

Each of these instructions takes an 8-bit quarter of the input of F, reads the according S-Box value (see section 4) from the memory and processes either an *xor* or an *add* instructions on this value. It has to be mentioned that each of the $S_{i,j}$-operators comprises an addition of an offset j to a base address S_i and reads the value of the computed address from memory. By calling these instructions in a sequence, we gained a first approach to support Blowfish by dedicated instructions. However, memory accesses and additions consume lots of computation time and therefore our instructions did not meet the cycle length constraint given by the MIPS architecture. Furthermore, the reusability of the instructions, with respect to other block cipher algorithms is also very limited.

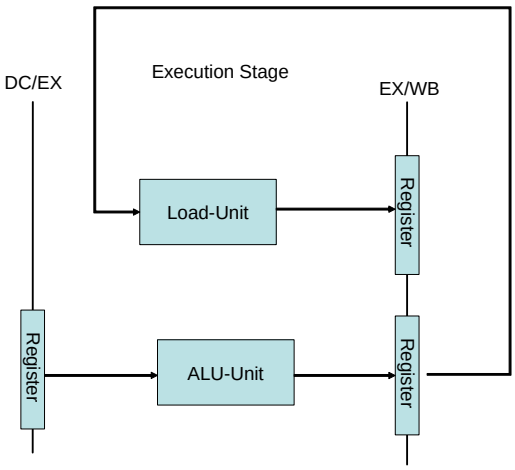

Fig. 6. Parallel S-Box access in the execution stage.

Refining our first approach, we decided to separate the core S-Box access from the remaining operations and to put it into a dedicated hardware unit. This unit was placed into the execution stage, such that it can be executed in parallel to other operations (fig. 6). As a consequence, the memory latencies related to S-Box accesses are completely hidden in the execution of the encryption instructions and do not affect system performance. We modified the encryption instruction putting focus on the number of additions in each of them. The result was that we developed four instructions, one for each S-Box, where each of them covers a S-Box access in the way that it calculates the address and pushes the result to the unit responsible for the pure S-Box memory access. Furthermore we developed an *add-xor* instruction and an *add-xor-xor* instruction to process the results from the S-Box accesses, so that we had now six instructions over all. This design made it possible to adapt our coprocessor speed to that of the MIPS architecture. In addition to this, the reusability for other block cipher algorithms is much better than in our first approach. For example, the *add-xor* instruction can be used as a pure *add* instruction just by setting one operand to zero.

5.3 Architecture Implementation

At the last stage of the design flow, the *architecture implementation phase* (fig. 2 in section 1), the ADL architecture description is used to generate an architecture description on *Register Transfer Level*(RTL). The *RTL-Processor-Synthesis* can be triggered to generate synthesizable HDL-code for the complete architecture. Numbers on hardware cost and performance parameters (e.g. design area, timing) can be derived by running the HDL processor model through the standard logic synthesis flow. On this high level of detail the designer can tweak the computational efficiency of the architecture by experimenting with different implementations of the data path.

First synthesis results from the architecture defined in exploration phase 3 showed the potential for area improvement in both the pipeline and the general purpose register file. In order to reduce chip size, we applied two more optimizations of the coprocessor.
In the first implementation loop, we removed all unnecessary and redundant functionality like *shift*, *mul* or *add* operations. For example, having an *add-xor* instruction made original *add* instructions redundant. Furthermore, we enhanced the architecture by *increment* and *decrement* operations. These operations can be used for the processing of loop counters, instead of using 32-bit *adders*.

In the second implementation loop, we reduced the number of general purpose registers from 15 down to 9. Additionally, the number of ports of the general purpose register file was reduced. The remaining coprocessor architecture just consists of instructions for memory access, register-copy, increment, decrement and xor. Along with these, 6 dedicated instructions for symmetric encryption as well as 9 general purpose and 3 special purpose registers to hold the S-Box values

were implemented. This architecture is sufficient to comfortably implement an encryption, decryption and a key generation function for symmetric block cipher algorithms.

5.4 Experimental Results

The architecture parameters considered for our design decisions during exploration phase were code size and number of cycles. During the implementation phase, chip area and timing were taken into account. In tables 1 and 2, the processed iterations in *exploration phase* are numbered from *exploration 1* to *exploration 3* and from *synthesis 1* to *synthesis 3* in the *implementation phase*.

	simulation results		
	exploration 1: (standalone simulation)	**exploration 2: (first cop. approach)**	**exploration 3: (second cop. approach)**
code size	531	235	267
number of cycles	917844	117546	176319

Table 1. Simulation results in the architecture exploration phase.

	area consumption (kGates)		
architecture part	**synthesis 1: (extended by encryption instructions)**	**synthesis 2: (eliminated redundant instructions)**	**synthesis 3: (with reduced register ports)**
total	31.4	25.8	22.2
pipeline	21.1	15.0	14.9
register file	10.1	10.5	7.1

Table 2. Area Consumption in the architecture implementation phase.

Table 1 shows that the employment of our coprocessor of *exploration 3* results in an overall speed-up of the Blowfish encryption algorithm by a factor of five.
Although the number of instructions of *exploration 2* is less than the corresponding number of *exploration 3* it turned out, that the timing constraint specified by the MIPS, cannot be met by the model *exploration 2*, whereas our final model of the *exploration phase* showed an equivalent timing to the MIPS processor.
Table 2 confirms our statements from section 5.3. The first synthesis resulted in a core which had an area consumption of 31.4 kGates. We were able to reduce this

area size to 22.2 kGates. In the first implementation loop, the area consumption of the pipeline was reduced from 21.1 kGates down to 15.0 kGates. Furthermore, in the second implementation loop, we decreased the area of the register file by 3.4 kGates down to 7.1 kGates.

6 Conclusions

In this paper we illustrated the successive refinement flow of processor architectures within an architecture exploration procedure. In our case study we used an *IPv6* protocol stack implementation developed by Microsoft Research which we combined with the Blowfish block cipher algorithm as the *IPsec* encryption. We have designed a coprocessor for efficient implementation of symmetric block cipher algorithms by providing an application specific instruction set. To access the encryption procedures from the C-level, we inserted dedicated intrinsic functionality into the generated MIPS C-compiler. By using the whole LISA tool suite for our case study, we were able to show that overall IP processing can be very efficiently supported by a small set of hardware accelerations without loss of flexibility due to future developments in the area of symmetric encryption. Because of the very simple and common structure of the Blowfish encryption algorithm, we believe that our approach can be adapted to other block cipher algorithms as well.

References

1. ACE – Associated Computer Experts bv. *The COSY Compiler Development System*
   http://www.ace.nl.
2. O. Y. H. Cheung, K. H. Tsoi, P. H. W. Leong, and M. P. Leong. Tradeoffs in Parallel and Serial Implementations of the International Data Encryption Standard (IDEA). In *Lecture Notes in Computer Science, vol. 2162*, 2001.
3. P. Chodowiec, P. Khuon, and K.Gaj. Fast Implementations of Secret-Key Block Ciphers Using Mixed Inner- and Outer-Round Pipelining. In *FPGA*, 2001.
4. CoWare Inc. *http://www.coware.com*.
5. R. Draves, B. Zill, and A. Mankin. Implementing IPv6 for Windows NT. In *Windows NT Symposium Seattle*, Aug. 1998.
6. A. Fauth, J. Van Praet, and M. Freericks. Describing Instruction Set Processors Using nML. In *Proc. of the European Design and Test Conference (ED & TC)*, Mar. 1995.
7. A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 1999.
8. A. Hoffmann, H. Meyr, and R. Leupers. *Architecture Exploration for Embedded Processors With Lisa*. Kluwer Academic Publishers, Jan. 2003. ISBN 1-4020-7338-0.

9. D. Lanner, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. Chess: Retargetable Code Generation for Embedded DSP Processors. In P. Marwedel and G. Goosens, editors, *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.
10. O. Mencer, M. Morf, and M. Flynn. Hardware Software Tridesign of Encryption for Mobile Communication Units. In *ASSP*, 1998.
11. M.Leong, O.Cheung, K. Tsoi, and P.Leong. A Bit-Serial Implementation of the International Data Encryption Algorithm (IDEA). In *IEEE Symposium on Field-Programmable Custom Computing Machhines*, 2000.
12. Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, and Heinrich Meyr. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. In *Proc. of the Design Automation Conference (DAC)*, Jun. 2002.
13. S. Kobayashi, Y. Takeuchi, A. Kitajima, M. Imai. Compiler Generation in PEAS-III: an ASIP Development System. In *Workshop on Software and Compilers for Embedded Processors (SCOPES)*, 2001.
14. M. Hohenauer H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, and H. Meyr. A Methodology and Tool Suite for C Compiler Generation from ADL Models. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 2004.
15. O. Schliebusch, M. Steinert, G. Braun, A. Nohl, R. Leupers, G. Ascheid, and H. Meyr. RTL Processor Synthesis for Architecture Exploration and Implementation. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 2004.
16. B. Schneier. *Applied Cryptography*. Addison-Wesley Publishing Company, Boston, Jun. 1996. ISBN 0-471-11709-9.
17. B. Schneier, J. Kelsey, D. Whiting, D. Wagner, and C. Hall. Twofish: A 128-Bit Block Cipher. Jun. 1998.
18. T. Grötker, S. Liao, G. Martin, S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.
19. Target Compiler Technologies. *CHESS/CHECKERS http://www.retarget.com*.
20. V. Rajesh and R. Moona. Processor Modeling for Hardware Software Codesign. In *Int. Conf. on VLSI Design*, Jan. 1999.
21. A. Wieferink, T. Kogel, R. Leupers, G. Ascheid, and H. Meyr. A System Level Processor/Communication Co-Exploration Methodology for Multi-Processor System-on-Chip Platforms. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 2004.