Impact of Configurability and Extensibility on IPSec Protocol Execution on Embedded Processors

Nachiketh R. Potlapally[†], Srivaths Ravi[‡], Anand Raghunathan[‡], Ruby B. Lee[†], and Niraj K. Jha[†]
[†] Dept. of Electrical Engineering, Princeton University, Princeton, NJ 08544
[‡]NEC Laboratories America, Princeton, NJ 08544
{*npotlapa*, *rblee*, *jha*}@princeton.edu, {*sravi*, *anand*}@nec-labs.com

Abstract - *Security protocols, such as IPSec and SSL, are being increasingly deployed in the context of networked embedded systems. The resource-constrained nature of embedded systems and, in particular, the modest capabilities of embedded processors make it challenging to achieve satisfactory performance while executing security protocols.*

A promising approach for improving performance in embedded systems is to use application-specific instruction set processors that are designed based on configurable and extensible processors. In this work, we perform a comprehensive performance analysis of the IPSec protocol on a state-of-the-art configurable and extensible embedded processor (Xtensa from Tensilica, Inc.). We present performance profiles of a lightweight embedded IPSec implementation running on the Xtensa processor, and examine in detail the various factors that contribute to the processing latencies, including cryptographic and protocol processing. In order to improve the efficiency of IPSec processing on embedded devices, we then study the impact of customizing an embedded processor by synergistically (a) configuring architectural parameters, such as instruction and data cache sizes, processor-memory interface width, write buffers, etc., and (b) extending the base instruction set of the processor using custom instructions for both cryptographic and protocol processing. Our experimental results demonstrate that upto 6X speedup in IPSec processing is possible over a popular embedded IPSec software implementation.

Keywords: *Configurability, Embedded Processors, Embedded Security, Embedded Systems, Extensibility, IPSec, Performance, Security Protocols.*

I. Introduction

Embedded systems are designed under a wide range of constraints, including cost, performance, and power consumption. Security has traditionally been an important consideration in the design of specific embedded systems, such as smart cards. As embedded systems are used in increasingly diverse applications to perform critical functions and access sensitive information, security has become a widespread concern in their design. Due to the networked nature of many modern embedded systems, they are exposed to the myriad security threats that we have experienced with personal computers (PCs) and the Internet.

Conventional "functional" security measures, such as cryptographic algorithms, secure communication protocols, and secure computation and storage mechanisms, can also be applied to embedded systems. However, various design constraints and usage scenarios that are unique to embedded systems usher in new challenges. For example, due to cost and power constraints, many embedded systems do not possess the computing power of the general-purpose microprocessors used in PCs or workstations. This leads to a "security processing gap," or a disparity between the computational requirements for security and the capabilities of embedded processors. For example, performing 3DES symmetric encryption plus SHA-1 hashing at a data rate of 2Mbps requires a computational capability of around 130 Million Instructions per Second (MIPS) [1], which is beyond the capabilities of many low-end embedded processors.

A promising architectural approach to obtain satisfactory performance with low cost and low power is to use processors that are customized to the application or application domain targeted by the embedded system. Application-specific instruction set processors (ASIPs) combine the performance and power advantage of application-specific integrated circuits (ASICs) with the versatile programmability of general-purpose processors, without having to deal with the time-to-market issues inherent in custom ASIC design. While ASIPs offer a good tradeoff between flexibility and efficiency, the primary challenge to their adoption has been the complexity of designing a processor and its supporting compiler and software tool chain from scratch. *Configurable* and *extensible* processors have recently emerged as an effective way to design ASIPs by leveraging a pre-designed customizable base processor. Several established and emerging companies have developed embedded processors with configurable and extensible features [2]–[6]. Configurability refers to the ability to choose high-level architectural parameters, such as functional units (multiplier, accumulator, *etc.*), register file size, cache architecture, memory configuration, *etc.*, to best suit the needs of the target application. The provision for extending the instruction set of the configured processor by adding some special-purpose instructions is termed extensibility. A synergistic use of configurability and extensibility can give rise to ASIPs that are very efficient in satisfying the design constraints imposed by the end application.

In this work, we analyze the performance of the popular network security protocol, IPSec, executing on a commercial configurable and extensible embedded processor (Xtensa from Tensilica, Inc.). We investigate the performance tradeoffs resulting from architectural configurability and extensibility, and demonstrate how they can be explored to achieve high performance with minimal hardware complexity. We chose the IPSec protocol since it is relevant to networked embedded systems, which represent a major and rapidly growing part of the embedded systems market. IPSec is a network-layer security protocol, which provides for confidentiality and integrity of the communicated data. The key benefits of IPSec include transparency to applications (*i.e.*, applications can make use of the security services offered by IPsec without any changes), and flexibility (it can be used in different modes: end-to-end for securing traffic between two hosts, route-to-route for protecting a certain set of network links, or edge-to-edge for forming a secure tunnel between two trusted networks through an untrusted network).

A. Paper Contribution

Previous work aimed at improving security performance tended to target the acceleration of the cryptographic algorithm computations, ignoring the overhead of security protocol processing [7]–[9]. However, this approach does not suffice for security protocols

Acknowledgment: This work was supported by NSF under Grant No. CCR-0326372.

such as IPSec, which also contain a significant amount of non-cryptographic computation in the form of packet processing. Packet processing operations, like building and removing packet headers, looking up policy databases *etc.*, have been shown to be memory-intensive [10]. They pose a considerable bottleneck in IPSec operation as the required data rates increase. This trend will become worse as the gap between the processor and memory speeds widens in the future [11].

In this paper, we investigate the design space of a configurable and extensible embedded processor platform for improving the performance of IPSec-like security protocols. Our analysis identifies points in the design space that simultaneously improve the performance of both the compute-intensive cryptographic components, and the memory behavior of the protocol processing part, in a cost-effective manner on an embedded processor. *To the best of our knowledge, this is the first work to present a systematic analysis of IPSec execution on configurable and extensible processors.*

B. Related Work

Previous work dealing with enhancing processors (embedded and general-purpose) for faster security processing have focussed mainly on speeding up the performance of one or more cryptographic algorithms [7]–[9], [12]–[16] used in security protocols. In most scenarios, the processor is augmented with co-processors or with new custom instructions, specifically for speeding up cryptographic computations. Such cryptographic enhancements are increasingly being deployed in commercial embedded systems (*e.g.*, cell phone application processors, such as TI's OMAP 1610 [17] and NEC's MP211 [18], feature cryptographic hardware for accelerating symmetric ciphers and hashing algorithms).

Detailed performance analyses of security protocols, such as SSL, on embedded processors reveal that cryptographic computations form only a part of the overall workload [19] and, hence, cryptographic hardware accelerators alone will not suffice. For application-level protocols, such as SSL, tuning the operating system to the working of the application [20] is one possible way to address some of the non-cryptographic processing bottlenecks. In the case of IPSec running on a general-purpose computing system, it has been shown that processing overheads, such as network processing, data copying and transfer times, severely limit the utility of cryptographic hardware in certain scenarios (e.g., TCP bulk transfers) [21]. For low-end embedded devices, lightweight software implementations of IPSec [22] provide a good starting point for further optimizations.

Commercial hardware solutions for speeding up both cryptographic and packet processing in high-performance processors can be found in the enterprise server and backbone router markets [23]–[25]. These products integrate multiple specialized cores on a single chip to provide high-performance secure packet processing. A comparison of various techniques adopted for speeding up IPSec processing on high-performance processors can be found in [26]. However, the hardware costs of these solutions preclude them from being directly applicable in many embedded systems.

Our work targets the development of a high-performance IPSec solution for resource-constrained embedded devices. Towards this end, we first analyze the performance of a lightweight IPSec software implementation on a state-of-the-art embedded processor (Xtensa from Tensilica, Inc.). We then exploit the processor's configurability (micro-architectural parameters) and extensibility (custom instructions) to provide a holistic speedup to IPSec for both its cryptographic and protocol processing components.

The rest of this paper is organized as follows. Section II provides various performance statistics for IPSec processing, and motivates the opportunities for using configurability and extensibility to improve performance. Section III describes our performance analysis and architectural exploration framework, and studies various optimizations for accelerating IPSec on embedded processors. Section IV concludes this paper.

II. Motivation

In this section, we analyze the performance of IPSec, and motivate how the configurability and extensibility of an embedded processor can be used to improve it.

A. Cryptographic and Protocol Components of IPSec Processing

Fig. 1 breaks down the performance of IPSec processing on the client side of a secure client-server transaction. We separately consider the Authentication Header (AH) and Encapsulating Security Payload (ESP) modes of IPSec. The client is a web browser linked with embedded IPSec [22] and a lightweight network protocol stack (lwIP) [27], and is profiled on Tensilica's Xtensa T1050.1 embedded processor [2]. In these experiments, IPSec used the AES cipher for data encryption/decryption, and the MD5 algorithm for message authentication. Prior to invoking IPSec, pre-negotiated session keys were made available on both the client and server. Further details of our experimental framework are provided in Section III-A.

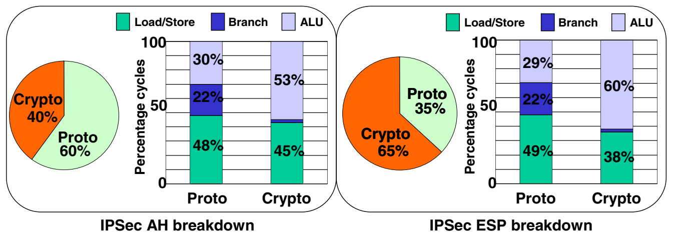


Fig. 1. Breakdown of IPSec processing into cryptographic and protocol components, and into various instruction categories

First, we measured the number of cycles spent by the client in IPSec-related protocol and cryptographic processing. The piecharts show that nearly 48% (averaged over multiple traces, and across both modes) of the IPSec processing cycles are spent in protocol processing. We further analyzed the cryptographic and protocol components of IPSec processing, and broke them down into cycles spent in executing different types of instructions. The results are shown in the bar graphs of Fig. 1. We can see that protocol and cryptographic processing have complementary behavior: protocol processing spends majority of its cycles in loads/stores and branch instructions, while cryptographic processing cycles are primarily spent in ALU instructions. This shows that cryptographic operations primarily comprise computational tasks, while protocol functions spend most of their cycles in memory transfers and issues arising out of transferring control of program execution. Thus, we observe that the cryptographic tasks are suited for optimization using extensibility, whereas the performance of protocol functions can be improved by judiciously selecting the values of the configurable parameters.

B. Functional Analysis of Protocol and Cryptographic Processing

From the above experiments, we also gathered performance statistics of individual functions involved in protocol and cryptographic processing. The results are summarized in Fig. 2.

The profile for protocol processing consists of four main components: building and stripping headers (collectively called *header processing*), loading values into security databases and looking them up later, checksumming, and data capture using the IPSec device driver. The first bar in Fig. 2 shows the percentage breakup of protocol processing cycles among these functions. Except for checksum, which involves arithmetic computations, the other functions are dominated by memory interactions. Hence, in our work, the checksum operation is targeted by adding custom instructions, while the other operations are optimized using configurability. The percentage cycle contribution of the constituent functions of AES (used in the ESP protocol) and MD5 (used in the AH protocol) are shown by the second and third bars in Fig. 2, respectively. All these functions involve intensive computations and, hence, can be sped up by custom instructions.

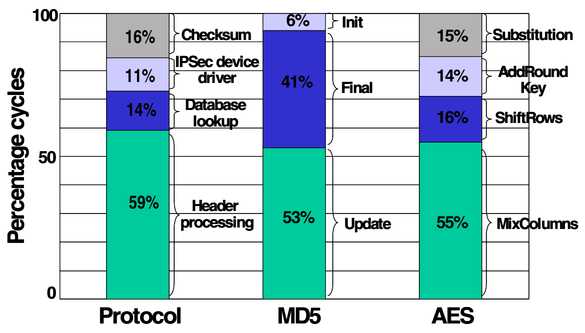


Fig. 2. Contributions of core functions in cryptographic and protocol components of IPSec processing

III. Architectural Exploration and Performance Analysis

In this section, we first describe the architectural exploration and performance analysis framework used in this work (Sections III-A and III-B). We then evaluate the impact of tuning various extensible and configurable parameters of an embedded processor on the performance of IPSec (Sections III-C and III-D). Finally, we discuss the implications of our experiments (Section III-E).

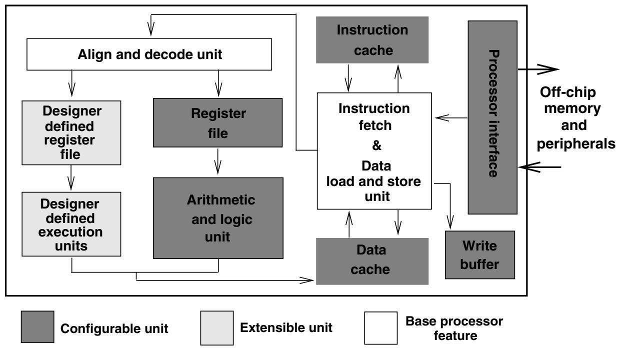
A. Architectural Design Space

Tensilica Xtensa is a high-performance embedded processor with a 32-bit reduced instruction set computer (RISC) architecture. It has a five-stage pipeline, and 32 general-purpose registers. It uses 16-bit and 24-bit instruction words for a denser encoding for reducing the processor code size. It employs register windows for faster handling of procedure calls. Fig. 3 shows the architectural features of the Xtensa processor partitioned into three categories. They include base instruction-set architecture (ISA) features that cannot be altered, configurable options, and extensible options. Configurable options allow for settings to the micro-architectural parameters, such as cache size, associativity, and line size, processor interface (PIF) width, *etc.* Extensible options allow the designer to extend the base ISA using special instructions that invoke custom application-specific functional units and registers integrated into the processor's datapath. The goal of our work is to find the configuration in this design space that would provide the maximum performance improvement to IPSec while satisfying the constraints typical of embedded systems.

B. IPSec Performance Analysis and Optimization Methodology

Fig. 4 describes the methodology used in this work. The objective of this methodology is to explore the configurable and extensible design space for the Xtensa processor described in the previous section, and improve the performance of IPSec running on the base Xtensa platform. The methodology consists of three phases: (i) gathering real-world IPSec traces[1] in a data collection phase, (ii) using the traces to analyze the performance of IPSec on the Xtensa processor through instruction-set simulation, and (iii) identifying

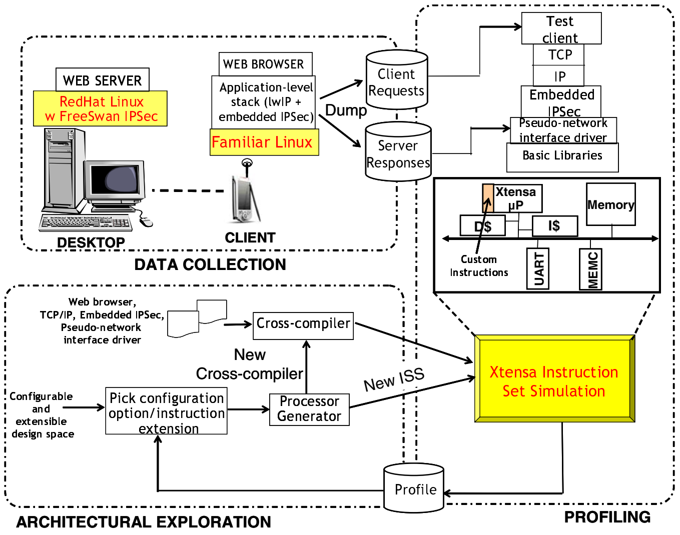
[1]Note that the trace files are a necessity since most instruction-set simulators (ISSs), including the Xtensa ISS, do not model the network interface, and hence, cannot support online simulation of client-server transactions.



Align and decode unit
Designer defined register file
Register file
Designer defined execution units
Arithmetic and logic unit
Instruction cache
Instruction fetch & Data load and store unit
Data cache
Processor interface
Write buffer
Off-chip memory and peripherals
Configurable unit
Extensible unit
Base processor feature

Fig. 3. Xtensa's architectural features

performance hotspots and tuning the processor by setting various configuration parameters and/or selecting custom instructions. The result of step (iii) is a new processor, which is then fed back to step (ii) to evaluate the performance impact of the modifications made.



WEB SERVER
RedHat Linux w FreeSwan IPSec
WEB BROWSER
Application-level stack (lwIP + embedded IPSec)
Familiar Linux
Dump
Client Requests
Server Responses
DESKTOP
CLIENT
DATA COLLECTION
Test client
TCP
IP
Embedded IPSec
Pseudo-network interface driver
Basic Libraries
Xtensa µP
Memory
D$
I$
Custom instructions
UART
MEMC
Web browser, TCP/IP, Embedded IPSec, Pseudo-network interface driver
Cross-compiler
New Cross-compiler
Configurable and extensible design space
Pick configuration option/instruction extension
Processor Generator
New ISS
Xtensa Instruction Set Simulation
Profile
ARCHITECTURAL EXPLORATION
PROFILING

Fig. 4. The three phases of our design methodology: data collection, performance profiling and architectural exploration

We now examine each phase in further detail:

- **Data collection**: We use an IPSec connection between a web server and a client connected over a local area network as the testbed for data collection. The web server consists of a 700MHz PC with 256MB RAM running Redhat Linux. It runs the FreeSwan IPSec software. The client used in the experiment is a Compaq iPAQ H3670 PDA, which contains an Intel SA-1110 StrongARM processor clocked at 206MHz. The client uses the Familiar distribution [28] of Linux as its operating system (OS), and runs the lightweight TCP/IP stack (lwIP) [27]. lwIP is a thin, application-level protocol stack written for small form-factor devices. An IPSec software targeted toward embedded devices, called embedded IPSec [22] is incorporated into lwIP to provide IPSec support on the client. The web browser program on the client initiates a connection to the server and sends it requests for web pages. Once a connection is established, the server supplies requested HTML pages to the client. The client requests and server responses are captured in trace files. Several trace files are captured for multiple IPSec-enabled web transactions having different traffic characteristics.
- **Profiling**: The client-side software used in data collection is cross-compiled to the Xtensa platform, and executed on

Xtensa's ISS. Since the ISS has no networking support, we introduce a pseudo-network interface driver to trap packets sent to the server and send appropriate responses from the captured trace files. By controlling certain parameters in the client, we can ensure that the sequence of packets exchanged remains unaltered (for example, certain values in the headers change from one transaction to the other, even though the same data are being exchanged). In this way, we can emulate the client-side operations on the ISS, and obtain IPSec's performance profile. The statistics gathered in this process include various measurements, *e.g.*, the number of processor cycles consumed by all the IPSec functions, instruction and data cache miss rates, memory penalty, *etc.*

- **Architectural exploration**: The performance profile derived in the previous step allows us to pick the configurable and extensible options relevant to this work. Each unique combination of options corresponds to an application-specific processor instance. For each processor instance, we used Xtensa's processor generator to generate a new RTL description of the processor and its corresponding software tool suite (cross-compiler and ISS). The profiling phase is then repeated to evaluate the impact of the selected options.

C. Tuning IPSec Using Extensible Options

In order to improve the performance of IPSec processing, compute-intensive operations in IPSec protocol and cryptographic processing are converted into special-purpose instructions. These compute-intensive operations (also called *hotspots*) are identified by studying the performance profile of IPSec protocol and cryptographic processing. Based on the performance analysis of IPSec presented in Section II, we infer the following about the hotspots in protocol and cryptographic processing components of the IPSec protocol.

- **Protocol processing**: From Fig. 2, we see that the *header processing* function consumes the most cycles. However, except for the *checksum* function, the protocol processing functions are dominated by memory transactions, and cannot be sped up with custom instructions. Thus, checksum is the only function in protocol processing which is suitable for custom instruction implementation.
- **Cryptographic processing**: Fig. 2 shows the functions which consume the majority of the cycles in MD5 (in the AH mode), and in AES (in the ESP mode), and therefore, can be targeted for custom instruction implementation to make cryptographic processing faster.

The following subsections present details of the custom instructions added to accelerate protocol and cryptographic processing in IPSec.

C.1 Checksum

The checksum operation is performed on data in every packet received. The RFC1071 [29] software implementation of the checksum operation accumulates 16-bit numbers into a 32-bit variable `sum`. The 32-bit sum in the accumulator is folded into a 16-bit result and complemented to get the final checksum value. The performance of checksum operation works out to 13 cycles/Byte on the base Xtensa processor.

The speed of the checksum operation is clearly limited by the serial addition of 16-bit data fetched from the memory. By introducing two 64-bit custom registers, we can provide new instructions that enable addition of multiple 16-bit numbers of the input data in parallel. We designed two custom instructions, ADD64() and FOLD64(). ADD64() implements the parallel addition of four 16-bit numbers with an accumulated sum, while FOLD64() performs the final fold operation. We use a load instruction to read in four 16-bit numbers from the memory into the custom register, and store instruction to write the final checksum result back to the memory. The performance of checksum improves to 9 cycles/Byte, providing a speed-up of 1.4X. The semantics of the instructions added are as follows:

- **ADD64(r1,r2)**: $r1[63:0] = r1[63:0] + r2[63:0]$.
- **FOLD64(r1,r2)**: $r2[15:0] = r1[63:48] + r1[47:32] + r1[31:16] + r1[15:0]$

C.2 AES

In AES encryption, the input data are divided into uniformly-sized input blocks of 128 bits, and a round transformation is applied multiple times to each input block [30]. An input block is referred to as the *state*, and is represented by a 4×4 matrix of bytes. The number of iterations of a round transformation applied on an input block is parameterized on two values: the input block size (128 bits) and the key length (128, 196 or 256 bits). Using the key as the input, a function is used to generate a different sub-key (called a *round key*) for each iteration of the round transformation. The round transformation is made up of the following operations,

- **Key addition**: Bytes of a round key (derived from the private key) are XORed with those of the state.
- **Substitution**: Each byte in the state is replaced by a byte in the S-box obtained by using the original byte as an index to perform the table lookup, *i.e.*, byte x is replaced by S[x].
- **Row shifting**: The rows of the state are cyclically shifted by fixed offsets.
- **Column mixing**: The columns of the state are considered to be polynomials over GF(2^8). They are multiplied by a polynomial, $3x^3 + x^2 + x + 3$ modulo $x^4 + 1$.

However, an optimization suited for 32-bit processors allows us to implement round transformation on a column of the state matrix as four table lookups and four XORs as shown below,

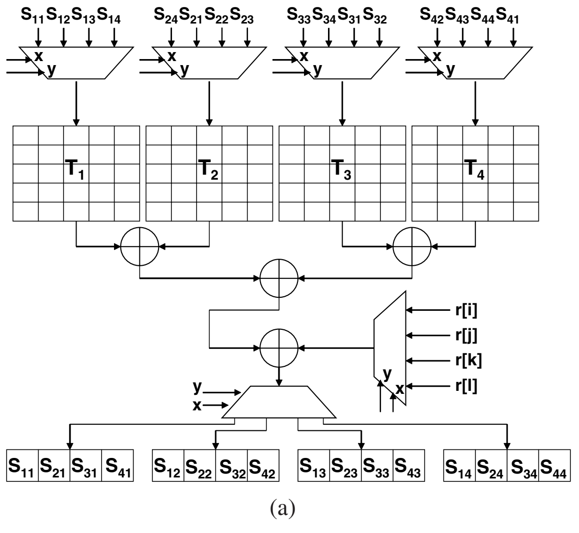
$$\begin{bmatrix} e_{1,j} \\ e_{2,j} \\ e_{3,j} \\ e_{4,j} \end{bmatrix} = T_1[a_{1,j}] \oplus T_2[a_{2,j}] \oplus T_3[a_{3,j}] \oplus T_4[a_{4,j}] \oplus \begin{bmatrix} k_{1,j} \\ k_{2,j} \\ k_{3,j} \\ k_{4,j} \end{bmatrix}$$

where, $a_{i,j}$ are the bytes in the jth column of the state, $k_{i,j}$ are the bytes of the round key, and $e_{i,j}$ are the bytes of the result. T_1, T_2, T_3, and T_4 are the four lookup tables, each having 256 entries of four-byte words. Thus, encryption round transformation on the entire *state* is realized using 16 table lookups and 16 XOR operations. This optimization greatly speeds up the rate of AES encryption.

We introduce four hardware tables filled with pre-computed values, and a custom instruction to implement the round transformation. Fig. 5(a) shows the functional view of the custom instruction. The custom instruction takes the state as an input, and operates on it column-by-column. S_{ij} indicates the byte in the ith row and jth column of the state. The custom instruction is multi-cycled, and processes one column of a state per cycle. In each cycle, select signals of the multiplexers (x, y) are set to appropriate values, in order to select bytes of the correct column. Outputs of table lookups are XORed with each other, and the result is XORed with the round key (variable r in the figure). Thus, the state is processed in four processor cycles. A similar instruction, with its own set of four hardware tables, is implemented for decryption. In addition, load and store instructions are defined to read in data and write out results.

The semantics of the instructions added to speed up AES execution are as follows:

- **ENC_ROUND(S,r,i,j,k,l)**:
  $S_{i1} = r[i] \oplus T1[S_{11}] \oplus T2[S_{24}] \oplus T3[S_{33}] \oplus T4[S_{42}]$,
  $S_{i2} = r[j] \oplus T1[S_{12}] \oplus T2[S_{21}] \oplus T3[S_{34}] \oplus T4[S_{43}]$,
  $S_{i3} = r[k] \oplus T1[S_{13}] \oplus T2[S_{22}] \oplus T3[S_{31}] \oplus T4[S_{44}]$,
  $S_{i4} = r[l] \oplus T1[S_{14}] \oplus T2[S_{23}] \oplus T3[S_{32}] \oplus T4[S_{41}]$
  (where, $S_{i1}, S_{i2}, S_{i3}, S_{i4}$ are the first, second, third and fourth columns of the state, and $T1$, $T2$, $T3$ and $T4$ are hardware tables).



$S_{11}S_{12}S_{13}S_{14}$
$S_{24}S_{21}S_{22}S_{23}$
$S_{33}S_{34}S_{31}S_{32}$
$S_{42}S_{43}S_{44}S_{41}$
x
y
T_1
T_2
T_3
T_4
r[i]
r[j]
r[k]
r[l]
S_{11} S_{21} S_{31} S_{41}
S_{12} S_{22} S_{32} S_{42}
S_{13} S_{23} S_{33} S_{43}
S_{14} S_{24} S_{34} S_{44}
(a)

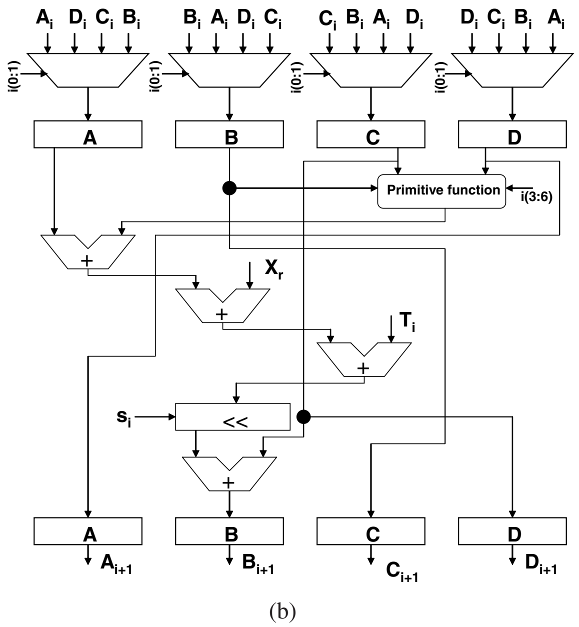


Fig. 5. Functional view of custom instructions (a) ENC_ROUND in AES, and (b) MD5_ITERATION in MD5

- **DEC_ROUND(S,r,i,j,k,l)**: It has a structure similar to *ENC_ROUND*().

The performance of the software implementation of AES is 415 cycles/Byte, while the use of custom instructions improves the performance to 102 cycles/Byte. Thus, the addition of custom instructions provides a speed-up of 4.1X in AES encryption.

C.3 MD5

MD5 takes input data of arbitrary length, and outputs a fixed-length message digest of 128 bits [31]. It breaks up the input data into blocks of 512 bits each, and processes each block sequentially using a compression function. At each step, the compression function takes two inputs, a 512-bit input block and a 128-bit block (output by the application of compression on the previous 512-bit input block), and outputs a 128-bit block. The 128-bit block obtained after compressing the last 512-bit input block is the output message digest. The compression function comprises four rounds, each having a similar structure, but parameterized on different primitive sub-functions. Each round consists of 16 iterations of its corresponding functionality implemented using the primitive function. Each iteration is parameterized on a 32-bit constant value. The 128-bit input block is broken into four 32-bit words, A, B, C, and D, which are successively modified by each iteration. The primitive functions used in the four rounds are given by $(x \wedge y) \vee (\overline{x} \wedge z)$, $(x \wedge z) \vee (y \wedge \overline{z})$, $x \oplus y \oplus z$, and $y \oplus (x \vee \overline{z})$, respectively, where x, y, and z are 32-bit inputs to the primitive functions.

According to the performance profile of MD5 shown in Fig. 2, *Final* and *Update* functions dominate its computation. The compression function described above forms the computational kernel of both these functions, and therefore, we target the compression function for custom instruction addition. We define a custom instruction MD5_ITERATION that implements any iteration from among the 4×16 iterations comprising the compression function. MD5_ITERATION implements the functionality shown in Fig. 5(b). The semantics of this instruction are as follows:

- **MD5_ITERATION**$(i, A_i, B_i, C_i, D_i, X_r, T_i, s_i)$: $(A_{i+1}, B_{i+1}, C_{i+1}, D_{i+1}) = F_i(i, A_i, B_i, C_i, D_i, X_r, T_i, s_i)$, where F_i is the function performed by the circuit shown in Fig. 5(b).

Each iteration takes as its input the current message digest state, as given by A_i, B_i, C_i and D_i, a 32-bit subset (X_r) of the 512-bit input block, and a constant value, T_i, corresponding to that iteration, chosen from a table of pre-computed constants. In each round, the appropriate pre-defined non-linear primitive function g is used. Each iteration outputs updated values A_{i+1}, B_{i+1}, C_{i+1}, D_{i+1}, which are passed on to the next iteration. A load instruction is defined to read in words of the 512-bit input block.

The performance of the software implementation of MD5 is 105 cycles/Byte, while the use of custom instructions improves the performance to 34 cycles/Byte (a speed-up of 3.1X).

D. Impact of Configurable Options

In this section, we present the results of varying the configurable parameters in the presence of the extensible options described in the previous subsection and study their effect on IPSec processing. The values of the configurable parameters include (i) instruction and data cache parameters such as size (1KB, 2KB, 4KB, 8KB, 16KB and 32KB), line width (16B, 32B and 64B), and associativity (1-way, 2-way and 4-way), (ii) register file size (32 and 64), (iii) write buffer slots (4, 16 and 32), and (iv) PIF width (32-bit, 64-bit and 128-bit). In each case, the performance is measured in terms of number of cycles consumed *per packet* for protocol processing, encryption using AES in ESP, and hashing with MD5 in AH averaged over multiple workload traces. We use the **base processor configuration** defined as *1KB direct-mapped instruction and data caches with 16 Byte lines, 32 registers, 4 write buffer slots, and 32-bit PIF width*, as the basis for evaluating the effect of varying the configurable parameters. The results of our investigation are presented below:

- *Cache size*: Increasing I-cache size benefits both cryptographic and protocol processing. Improvements in the performance of AES and MD5 level off after 4KB. This happens because custom instructions reduce their code sizes by appreciable amounts (40% and 30%, respectively), and most of the reduced code probably fits within a 4KB I-cache. Larger D-cache size improves the performance of protocol processing, but, not that of cryptographic processing. AES and MD5 employ table lookups of sizes 8 KB and 2 KB in their operation, respectively. However, they are realized as hardware tables in the custom instructions, and thus not affected by the D-cache size.
- *Cache associativity*: Increasing I-cache associativity does not appreciably improve IPSec performance (both cryptographic and protocol processing components). Similar observation is true for D-cache associativity except in the case of protocol processing (whose performance improves by 10%). This indicates the presence of data conflicts in protocol processing.

- *Cache line width*: Longer I-cache line width benefits protocol processing and MD5 more than AES. This could be explained by the presence of longer basic blocks in MD5 and protocol processing code compared to AES. Thus, a longer line width better exploits the greater spatial locality in the former two. Longer D-cache line widths improve the performance of protocol processing alone. Protocol processing involves operations on packets comprising sequential bytes of data (hence, greater spatial locality).
- *Register file size*: Increasing the register file size from 32 to 64 drastically reduces the window overflows and underflows. However, overall improvements in IPSec processing are modest.
- *Write buffer depth*: Protocol processing spends about 30% of its cycles waiting for free slots in the write buffer. Thus, increasing the write buffer slots improves protocol processing by 10%. This has no effect on cryptographic processing.
- *PIF width*: Increasing PIF width to 128 bits improves cryptographic processing by 15%, and protocol processing by nearly 30%. Greater PIF width enables faster processing of memory misses. Since protocol processing is more memory intensive than cryptographic processing, it is benefitted by a greater amount.

E. Summary

In the previous two sub-sections, we investigated the effect of configurable and extensible parameters on IPSec-related protocol processing, MD5 (AH) and AES (ESP). As part of our investigation, the extensible parameters were implemented as custom instructions to replace the hotspots in IPSec-related protocol and cryptographic processing. With the extensible additions in place, we studied the configurable parameter values that would result in optimal performance. The configurable values identifed are *4KB, 4-way set-associative I-cache with 64-bit lines*; *4KB, direct-mapped D-cache with 64-bit lines*; *32 write buffer slots*, *register file with 32 registers*, and *128-bit PIF*.

TABLE I
BENEFIT OF EXTENSIBILITY AND CONFIGURABILITY

Option	Protocol	MD5	AES
Configurable	50 %	15 %	20 %
Extensible	5 %	65 %	75 %

Table I gives a summary of the amount by which IPSec-related protocol and cryptographic processing benefit from extensibility and configurability. Protocol processing benefits much more from configurability than extensibility. The opposite holds true for cryptographic processing. This is intuitive, since MD5 and AES are dominated by computations which can be suitably targeted by introduction of custom instructions, whereas protocol processing is dominated by memory-related issues which are best addressed through configurability rather than extensibility. Together, the optimal configurable parameters and extensible options improve IPSec processing in the AH and ESP modes by factors of 4X and 6X, respectively.

IV. CONCLUSIONS

In this paper, we studied how the configurable and extensible parameters of a state-of-the-art embedded processor affect the performance of IPSec running on it. Using a detailed analysis framework, we identified values for the configurable parameters, and introduced custom instructions to target compute-intensive IPSec tasks. The custom processor thus obtained provides a maximum speed-up of 6X for IPSec processing.

REFERENCES

[1] S. Ravi, A. Raghunathan, and N. Potlapally, "Securing wireless data: System architecture challenges," in *Proc. Int. Symp. System Synthesis*, pp. 195–200, Oct 2002.

[2] *Xtensa Application Specific Microprocessor Solutions - Overview Handbook*. Tensilica Inc. (http://www.tensilica.com), 2001.

[3] *ARCtangent^{TM}processor*. Arc International (http://www.arc.com).

[4] PICO Flex, Synfora Inc. (http://www.synfora.com).

[5] *ARM processors*. http://www.arm.com.

[6] *MIPS32^{TM}M4K^{TM}core*. MIPS Technologies (http://www.mips.com).

[7] Z. Shi and R. Lee, "Bit permutation instructions for accelerating software cryptography," in *Proc. Int. Conf. Application-specific Systems, Architectures & Processors*, pp. 138–148, July 2000.

[8] J. Burke, J. McDonald, and T. Austin, "Architectural support for fast symmetric-key cryptography," in *Proc. Int. Conf. Arch. Support for Prog. Lang. & Operating Systems*, pp. 178–189, Nov. 2000.

[9] S. Ravi, A. Raghunathan, N. Potlapally, and M. Shankaradass, "System design methodologies for wireless security processing platform," in *Proc. Design Automation Conf.*, pp. 777–782, June 2002.

[10] T. Blackwell, "Speeding up protocols for small messages," in *Proc. Special Interest Group on Data Comm.*, pp. 85–95, 1996.

[11] W. Wulf and S. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Comp. Arch. News*, pp. 20–25, Mar. 1995.

[12] L. Wu, C. Weaver, and T. Austin, "Cryptomaniac: A fast flexible architecture for secure communication," in *Proc. Int. Conf. Comp. Architecture*, pp. 110–119, June 2001.

[13] A. Hodjat, P. Schaumont, and I. Verbauwhede, "Architectural design features of a programmable high throughput AES coprocessor," in *Proc. Int. Conf. Information Technology: Coding and Computing*, pp. 498–502, Apr. 2004.

[14] M. McLoone and J. V. McCanny, "A single-chip IPSec cryptographic processor," in *Proc. Int. Wkshp. Signal Processing Systems*, pp. 133–138, Oct. 2002.

[15] D. Carlson *et al.*, "A high performance SSL IPSec protocol aware security processor," in *Proc. Int. Solid-State Circuits Conf.*, pp. 142–143, Feb. 2003.

[16] H. W. Kim and S. Lee, "Design and implementation of a private and public key crypto processor and its application to a security system," *IEEE Trans. Consumer Electronics*, vol. 50, pp. 214–224, Feb. 2004.

[17] *OMAP 1610 Platform*. Texas Instruments Inc. (`http://www.ti.com`).

[18] *MP211 Application Processor*. NEC Electronics (`http://www.necel.com`).

[19] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," *ACM Trans. Embedded Computing Systems*, vol. 3, pp. 461–491, Aug. 2004.

[20] M. Burnside and A. Keromytis, "Accelerating application-level security protocols," in *Proc. Int. Conf. Networks*, pp. 313–318, Oct. 2003.

[21] S. Miltchev, S. Ioannidis, and A. Keromytis, "A study of the relative costs of network security protocols," in *Proc. USENIX Annual Technical Conf.*, pp. 41–48, June 2002.

[22] *Lightweight IPSec Implementation*. (`http://www.hta-bi.bfh.ch/Projects/ipsec/`), 2004.

[23] *Hifn 7851 security processors*. Hifn Inc (`http://www.hifn.com/products/7851.html`), 2004.

[24] *NITROX security processors*. Cavium Inc (`http://www.cavium.com/processor_security.html`), 2004.

[25] *Intel IXP2850 network processor*. Intel Corporation (`http://www.intel.com/design/network/products/npfamily/ixp2850.htm`), 2004.

[26] R. Friend, "Making the gigabit IPSec VPN architecture secure," *IEEE Computer*, pp. 54–60, June 2004.

[27] *Lightweight TCP/IP Stack*. (`http://savannah.nongnu.org/projects/lwip/`), 2003.

[28] *Familiar Project*. http://familiar.handhelds.org.

[29] R. Braden, D. Borman and C. Partridge, *RFC1071 - Computing the Internet Checksum*. Network Working Group.

[30] J. Daeman and V. Rijmen, "Rijndael: The advanced encryption standard," *Dr. Dobb's Journal*, vol. 26, no. 9, pp. 137–139, 2001.

[31] W. Stallings, *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 1998.