Instruction Set Customization of Application Specific Processors for Network Processing : A Case Study

Mohammad Mostafizur Rahman Mozumdar, Kingshuk Karuri, Anupam Chattopadhyay,
Stefan Kraemer, Hanno Scharwaechter, Heinrich Meyr, Gerd Ascheid, Rainer Leupers

Institute for Integrated Signal Processing Systems
RWTH Aachen University, Germany

Abstract

The growth of the Internet in the last decade has made current networking applications immensely complex. Systems running such applications need special architectural support to meet the tight constraints of power and performance. This paper presents a case study of architecture exploration and optimization of an Application Specific Instruction set Processor (ASIP) for networking applications. The case study particularly focuses on the effects of instruction set customization for applications from different layers of the protocol stack. Using a state-of-the-art VLIW processor as the starting template, and Architecture Description Language (ADL) based architecture exploration tools, this case study suggests possible instruction set and architectural modifications that can speed-up some networking applications upto 6.8 times. Moreover, this paper also shows that there exist very few similarities between diverse networking applications. Our results suggest that, it is extremely difficult to have a common set of architectural features for efficient network protocol processing and, ASIPs with specialized instruction sets can become viable solutions for such an application domain.[1]

1 Introduction

In the last decade, the exponential growth of the Internet has made networking immensely complex and diverse. Modern network interfaces have to support high bandwidth requirements while ensuring secure and reliable services over unreliable communication channels. At the same time, the advent of portable and mobile communication gadgets, and their high connectivity to the Internet has made it necessary to provide networking services on small, embedded System-on-Chips (SoCs). General purpose computing engines are particularly unfit to handle network computing on such embedded platforms under tight area and energy consumption budgets. The need to find *application specific* alternative architectures for such systems is, therefore, quite urgent.

As an application area, network processing has already received a lot of attention from computer architects and system designers[13][15][14]. However, most of this attention has been directed to find optimized processor architecture for highly specialized tasks, such as routing, that can be generally performed on high-performance, special purpose computing engines like routers or web servers. Such architectures are usually designed to deliver very high throughput at the expense of high energy consumption, and therefore, not very suitable for embedded computing domain. In this context, customized ASIP based systems can prove to be viable solutions. An ASIP, as the name suggests, is a programmable processor customized for a particular application domain. The application specific features help ASIPs to achieve performance close to custom-built ASICs while retaining flexibility and programmability.

The Instruction Set Architecture (ISA) of an ASIP is usually customized with *special purpose instructions* to implement frequently executed cluster of operations in hardware. As recent studies show, careful selection of such instructions can easily speed-up execution enormously at a very low area overhead[1][2][3]. Unfortunately, in stark contrast to other application domains such as signal processing or media processing, very few studies have been done to examine the possibilities and effects of specialized instruction sets for networking applications as a whole.

This paper presents a case study of *instruction set customization for network processing applications*. The primary goal of this work has been to study networking applications from different layers of the TCP/IP [21] protocol stack, and to identify the possibilities of instruction set modifications based on our analysis. To properly quantify the advantages of different custom instructions w. r. t. area and performance, we have performed extensive benchmarking using a real-life VLIW processor[20] already optimized for media processing. Cycle count and area estimates, with and without special instructions, have been obtained using retargetable software tooling available with a state-of-the-art, ADL based processor design platform[19].

The major contribution of this paper is to analyze a variety of networking kernels - diverse in both functionality and computing requirements, and to study the similarities (or lack of that) between them to suggest a common set of ISA enhancements. Based on such analysis, this paper suggests instruction set and architectural modifications that can speed-up *some* networking applications significantly. Additionally, this paper shows that there exist very few common features be-

[1]This work has been supported by Philips Research, Eindhoven, Netherlands.

tween different networking tasks such as packet processing, error detection/correction or encryption, and, therefore, each of these sub-domains require separate attention for ISA customization.

This paper is organized as follows. After a discussion of the related work in section 2, the target architecture and the processor design platform used for this work are described in section 3. In section 4, we concentrate on the custom instruction identification and evaluation methodology. Section 5 presents the different custom instructions identified for various benchmarks, and a thorough analysis of their processing requirements. In section 6, we present and analyze the results of our benchmarking procedure, and finally, summarize the current work and present some future directions in section 7.

2 Related Work

Most modern networking models such as the OSI or TCP/IP are organized in several protocol layers. Each layer uses services of layers below it, and in turn, provides services to layers above it. For example, in the TCP/IP protocol stack[21], the TCP layer uses unreliable packet transfer facilities of the IP layer, and provides reliable, connection oriented services to the application layer. Detailed description of such layered architectures and their computational and resource requirements can be found in [22].

The processing requirements of different layers is extremely diverse, and varies from handling of electrical signals and raw bit-streams at the physical and data link layer, to providing high level web-content and data-security at the application layer. A summary of these varied processing requirements and the different processor architectures targeted to handle them can be found in [9]. Most of these current network-processors employ either massively parallel processing capabilities (e.g. Intel IXP 1200[14] or IBM PowerNP [13]), or uses expensive special purpose hardware. Moreover, such processing engines are mostly designed to provide high-throughput for very specialized tasks such as routing, traffic management etc. Such tasks are usually performed on dedicated routers or servers and are not meant to be run on end-user devices. So far, the relatively inexpensive option of employing special instructions for such tasks, as presented in this paper, has not been explored in great detail.

One area of network processing which has become a separate domain in its own right is *network security*. The heart of network security is cryptography. Therefore, significant amount of work has been devoted to design efficient processor architectures for cryptographic algorithms. In [3], an approach to optimize processor architecture by introducing local memory for cryptographic algorithms, such as the Advanced Encryption Standard (AES), is presented. This paper suggests that, by introducing small hardware tables and architecturally-visible state registers, it is possible to get 250% to 500% speed-ups. In [5], a case study of ASIP architecture exploration for efficient IPSec encryption is described. The target application selected is *Blowfish* block cipher algorithm. By using a dedicated ASIP co-processor, this case study reports a speed-up of around 500% for the computational core of Blowfish. In [6], an architectural analysis of cryptographic applications has been described. This analysis takes into account various architectural features like instruction set characteristics, Instruction Level Parallelism (ILP) and cache performance. In [7], Todd Austin's group have investigated architectural support for fast symmetric-key cryptographic algorithms like *3DES, Blowfish, IDEA, Mars, RC4, RC6, Rijndael* and *twofish*. By introducing new instructions and special caches into the architecture, this study reports a four fold speed-up.

This paper takes a look at general characteristics of network processing algorithms. From this view, it differs significantly from prior researches which concentrated on fragmented aspects of network processing such as cryptography or packet processing. Additionally, we only explore the effects of instruction set customization which comes with modest overhead in terms of area and energy consumption, and leave aside other architectural features such as ILP. For this purpose, we have selected an initial architecture that already supports a high-degree of ILP and has a highly optimized cache architecture. This helps us to eliminate special instructions with large amount of parallelism, and to select only those instructions that are un-affected by the ILP.

3 Target Architecture and Associated Tools

The evaluation of promising custom instructions requires benchmarking on a real-life architecture. For this purpose we chose Philips TriMedia [12] architecture. The frequent and incremental changes of the processor architecture for such work demands retargetable software tools which were generated using the LISATek [19] processor design platform. A brief overview of TriMedia and LISATek is given in this section so as to provide a better understanding of our work.

3.1 TriMedia Architecture

TriMedia32 (TM32), a member of Philips TM1x00 processor series with a VLIW core, has been designed for use in low cost, high performance video telephony devices [12]. TM32 is a five slot VLIW with 28 functional units offering a high degree of fine-grained parallelism. It has a regular RISC like instruction set and 128 *General Purpose Registers (GPRs)*. It is specially optimized for media and image processing to deliver enhanced audio and video quality, but there exists little support for networking.

One reason for selecting TM32 as our base architecture was its high ILP. Custom instructions with high amount of parallelism are bound to yield low speed-up values for such architectures, and are automatically eliminated from our consideration. Moreover, the primary area of our interest is network processing on embedded platforms. VLIWs are the common multiple-issue processors employed in embedded systems domain. This made TM32 a good choice for our work.

3.2 LISATek Tools

The LISATek [19] Processor Designer (LPD), available from CoWare Inc, is based on LISA 2.0 ADL. LISA allows system designers to specify the instruction set and micro-architecture of a processor at a level of abstraction higher than that of the Hardware Description Languages (HDLs). From this abstract description, a set of software tools, such as compiler, assembler,

linker, loader and Instruction Set Simulator (ISS) can be automatically generated. The availability of such retargetable tools prompted us to use LPD, since we needed to make frequent and incremental architectural modifications on TriMedia.

4 Instruction Set Customization Work-flow

This section outlines our methodology for identification and evaluation of special purpose instructions for a set of given benchmark applications. The list of selected benchmarks and the identified custom instructions are described in the next section. Instruction set

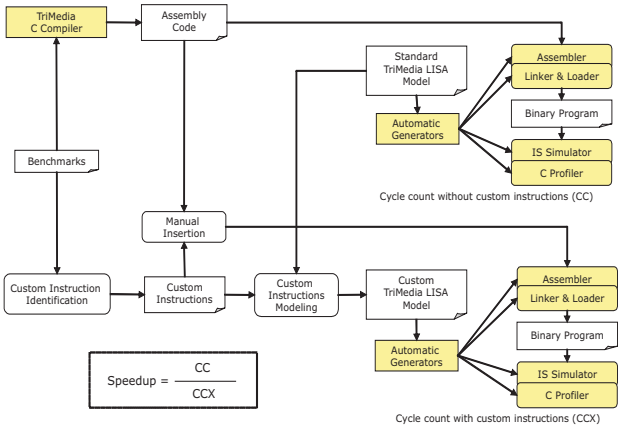

Figure 1. Instruction Set Customization Work-flow

extension depends on detecting clusters of operations in certain applications or application domains. Such a cluster of operations, when implemented as a single instruction, should improve execution speed and performance. Such a cluster of operations, forming a single instruction, is generally called a custom or a special-purpose instruction [8] [2]. Unfortunately, there exists no well established methodology on how to go about the task of identifying them. The custom instruction identification for the current work was done in the following steps:

- **Identification of hot-spots in the application :** According to Amdahl's law, any performance optimization must make the frequent case fast. This suggests that potential custom instructions are likely to be found in application *hot-spots*, i.e. heavily executed code regions. Such hot-spots were easily identified using application profiling.
- **Identification of custom instructions in hot-spots :** Potentially, any cluster of operations, i.e. any fragment of control or data flow graph inside the program hot-spots, can qualify for a custom instruction, provided it conforms to some user specified constraints. Such constraints usually depend on the target architecture. For example, the maximum number of inputs and outputs of instructions in the target architecture can be specified as a constraint. In such a scenario, the custom instruction identification problem reduces to a *constrained search* for control or data flow graph fragments that can maximize the execution performance. The problem is *NP-Hard* owing to the vastness of the search space.

  There exist efficient heuristics for automatically obtaining a reasonably good set of potential custom instructions for a given application [8]. Such automated identification procedure, however, is somewhat limited since it does not consider effects of compiler optimizations such as loop unrolling and eliminates custom instructions with internal memory accesses. Therefore, we decided to go for a mixed, semi-automatic approach. Firstly, we tried to identify custom instructions for each benchmark using an automated technique similar to [8]. At the same time, we analyzed the hot-spots manually to find candidate instructions better than those identified by automatic analysis. Finally, we combined the results of automatic and manual analysis by evaluating their impacts on performance.
- **Latency estimation of custom instructions:** Latency estimation of custom instructions was a difficult problem, since no information on the internals of TM32's hardware was known to us. To reach a reasonable estimate, we modeled each instruction in Verilog, and then synthesized it using Synopsys Design Compiler[24] with a 0.13 μm library. We obtained the minimum possible cycle length for each custom instruction by varying the synthesis constraints and then compared it to the minimum possible cycle length of the multiplication instruction (Since multiplication has the highest latency of the instructions modeled for our case study[3]). The latency for a Custom Instruction (CI) was then estimated using the following formula:

$$Lat_{CI} = \lceil \frac{L_{CI}}{L_{Mult}/Lat_{Mult}} \rceil$$

  where L_{Mult}, Lat_{Mult} and L_{CI}, Lat_{CI} are the minimum cycle lengths and latencies for multiplication and CI, respectively.
- **Short-listing of custom instructions based on simulation performance:** This was the final step of our instruction set customization process. It was done by modeling and simulating the TM32 architecture using LPD with and without the custom instructions, and eliminating those that offer small or no speed-ups.

The instruction set customization work-flow is presented in figure 1. Each benchmark application was first compiled with Philips TM32 compiler to obtain scheduled assembly code for the *standard* TM32 processor. The scheduled code was assembled and run on the *cycle accurate* ISS generated from the standard TM32 described in LISA, and cycle counts were recorded. At the same time, each custom instruction for the benchmark was manually inserted in the TM32 assembly code. This manually modified code was simulated on the ISS of a *custom* TM32 model with the

[2] For the rest of this paper, the terms *custom instruction* and *special instruction* has been used interchangeably

[3] In TM32, floating point division has the largest latency. However, since none of the selected benchmarks had any division - floating point or otherwise - we decided not to model them.

custom instruction. The cycle counts obtained were used to characterize the speed-ups.

5 Benchmarks and Custom Instructions

The major goal of our work has been to analyze the ISA specialization requirements for network processing as a whole. For this reason, we focused primarily on areas of network processing that are either most computationally intensive, or very common. Moreover, we tried to avoid applications that require well known architectural features other than special purpose instructions. For example, we decided to leave out routing protocols from consideration since they require fast table lookup mechanisms for significant hardware acceleration. Additionally, we were handicapped by the lack of compiler support for the complex custom instructions, and had to select relatively simple *algorithmic cores* whose assembly code can be easily modified. With these facts in mind we zeroed on the following areas and benchmarks:

- **Error detection and correction.** This is probably one of the most common applications found in different network layers. We selected the well-known Cyclic Redundancy Check (CRC) [18] algorithm as a representative of this class.
- **Security and cryptography.** Possibly the most computationally intensive area of networking. From this domain, we selected two private-key cryptographic algorithms - *GOST* and *Blowfish* [23] - whose source codes are freely available.
- **Authentication, secure key exchange protocols, digital signatures etc.** This can be regarded as a sub-domain of network security. We selected publicly available *diffie-hellman (DH)*[11][17] key exchange algorithm from this class.
- **Packet header processing, fragmentation and reassembly.** A packet fragmentation algorithm, FRAG [16], was selected as a representative of this class.

Table 1. Identified Custom Instructions

Benchmark	Instruction Syntax
CRC	**reflect8** $rsrc_1 \rightarrow rdst$ **reflect32** $rsrc_1 \rightarrow rdst$
Blowfish	**addxoradd** $rsrc_1$, $rsrc_2$, $rsrc_3 \rightarrow rdst$ **lshiftbitand** $(pram)$ $rsrc_1 \rightarrow rdst$ **rshiftbitand** $(pram)$ $rsrc_1 \rightarrow rdst$ **saccess1** $(pram)$ $rsrc_1$, $rsrc_2 \rightarrow rdst$ **saccess2** $(pram)$ $rsrc_1$, $rsrc_2 \rightarrow rdst$ **bff** $rsrc_1 \rightarrow rdst$
GOST	**gostf** $rsrc_1$, $rsrc_2 \rightarrow rdst$
DH	**cimul** $rsrc_1$, $rsrc_2 \rightarrow rdst_1$, $rdst_2$
FRAG	**addwords** $rsrc_1 \rightarrow rdst$

The following subsections describe each algorithm and the corresponding custom instructions. Table 1 summarizes the benchmarks and the corresponding custom instruction syntaxes. The following convention has been used while describing the *syntax* of each custom instruction :

- $pram$: signifies 7 bit constant value in the instruction word.
- $rsrc_i$: signifies 32-bit source register operand. The subscript designates the relative position of the operand in the instruction word. Each $rsrc_i$ occupies 7 bits in instruction word.
- $rdst_i$: signifies 32-bit destination register operand. Similar to $rsrc_i$.

5.1 CRC

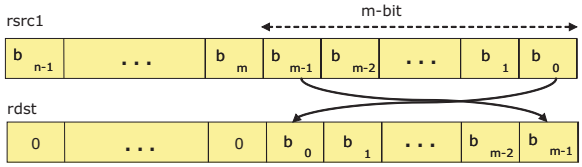

Figure 2. CRC Custom Instruction

CRC [18] is a frequently used algorithm for error detection. It converts a binary message to a polynomial and then divides it by a predefined *key* polynomial. The remainder of this division is CRC. At the transmitting end, the binary message and CRC are sent together. The receiver divides the message by the same *key* and flags an error if the remainder is *non-zero*.

Since automatic analysis of the CRC implementation failed to reveal any promising custom instruction, we decided to look into this application manually. This analysis indicated that a small function called *reflect* consumes most of the execution time. The semantics of an m bit reflect on n data bits is shown in figure 2. For the current CRC implementation, the reflection was always done on 32 bit data (i.e. $n = 32$) - either by 8 or 32 bits (i.e. $m = 8$ or 32). Therefore, we decided to design two custom instructions $reflect8$ and $reflect32$ for this application.

5.2 Blowfish

Blowfish[23] is a 64-bit block cipher with a variable-length key (up to 448 bits). The algorithm has two parts: *key expansion* and *data encryption*. In the key expansion phase, the variable-length key is used to generate a set of 32 bit sub-keys which are stored in arrays known as *S-Box*es and *P-Box*es. The encryption algorithm applies P-Box dependent *permutation* and S-Box dependent *substitution* on the plain text in 16 rounds. Like most other block-cipher algorithms, the heart of Blowfish is *S-Box based substitution* performed inside a function named F[4] (figure 3.(a)) that accounts for around 85% of the total execution time.

The F function causes significant amount of memory traffic due to the S-Box accesses. Since the automatic custom instruction identification algorithm eliminates custom instructions that make memory accesses *internally*, it could only identify the custom instructions *rshiftbitand, lshiftbitand* and *addxoradd*, shown in figure 3.(a) which failed to produce any significant speed-ups. Therefore, we decided to extend the *lshiftbitand*(to *saccess1*) and *rshiftbitand*(to *saccess2*) instructions with internal memory accesses as shown in figure 3.(b). The semantics of these instructions are shown in figure 4. Finally, as an experimental step, we

[4]Blowfish uses 4 S-Boxes each with 256 32-bit entries

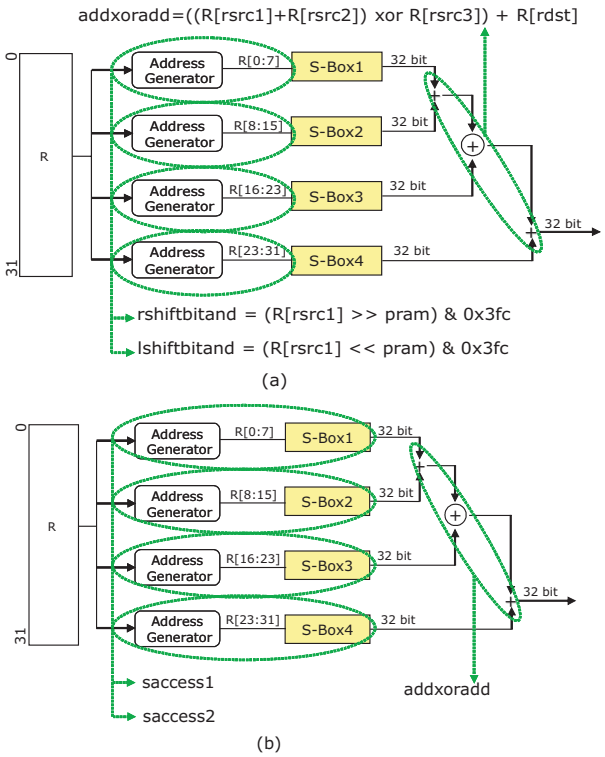

Figure 3. Blowfish *F* Function and Custom Instructions

also decided to incorporate the entire Blowfish *F* function into a single custom instruction *bff*. We evaluated two sets (one consisting of *saccess1, saccess2* and *addxoradd* and the other consisting of only *bff*) of custom instructions separately.

```
saccess1 (pram) rsrc1 , rsrc2 -> rdst
Semantics :
   unsigned int address;
   address = ((R[rsrc1] << pram) & 0x3fc);
   R[rdst] = MEM[address + R[rsrc2]];

saccess2 (pram) rsrc1, rsrc2 -> rdst
Semantics:
   unsigned int address;
   address = ((R[rsrc1] >> pram) & 0x3fc);
   R[rdst] = MEM[address + R[rsrc2]];
```

Figure 4. Semantics of *saccess1* and *saccess2*

5.3 GOST

GOST[10] is a 64-bit block cipher algorithm with a 256-bit key. Like Blowfish, GOST also uses S-Box substitution inside a similar *F* function. GOST uses 8 *S-Boxes* [5] and applies 32 encryption rounds on the plain text.

Like Blowfish, GOST also spends most of the execution time inside the *F* function. The semantics of the GOST *F* function is shown in figure 5. Since the automatic custom instruction identifier failed to return any good special instruction for GOST, we decided to

[5]Each GOST S-Box has 16 4-bit entries. But the implementation uses 4 arrays with 256 bytes each for representing them.

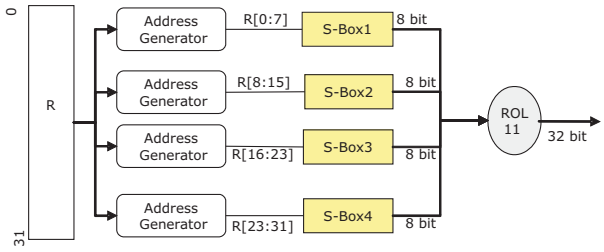

Figure 5. GOST *F* Function

implement the entire F function (named as *gostf*) as a custom instruction.

5.4 Diffie-Hellman (DH) Key Exchange Protocol

Diffie-Hellman (DH) is a secure key-exchange algorithm[11]. Using this algorithm, two communicating parties can decide on a *shared secret-key* over an *insecure communication channel*. Such a shared-key can be used later by private-key algorithms such as *Blowfish* or *GOST* to establish a secure communication channel between the two parties. Like many other public-key algorithms, DH uses modulus arithmetic quite heavily.

The analysis of the DH implementation[17] showed that the execution time is distributed in different functions i. e. there is no well defined hot-spot. Still, we decided to implement a special multiplication function named *cimul* (accounting for 34.85% of the total execution time), that produces a 64-bit output in two registers from two 32-bit inputs, as a custom instruction.

5.5 Frag

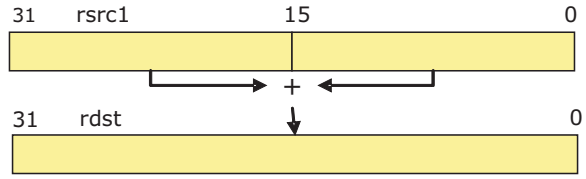

Figure 6. Frag Custom Instruction

FRAG is a packet fragmentation application [16] commonly found in routers - specially when the outgoing route has lower bandwidth than the incoming route. FRAG compares the length of every packet from the input channel to the maximum packet size handled by the outgoing route and fragments the packet if the length is larger than the maximum. This requires replication of the header and re-computation of the checksum for each fragment.

FRAG has three major functions dedicated to perform *fragmentation, checksum computation* and *memory copy*. This application is mostly memory access oriented, as it spends 50% time in the memory copy function. Rest of the time is spent in fragmentation (30%) and checksum (20%) function. FRAG doesn't have too many prominent patterns that can be implemented as custom instructions. The only custom instruction selected from FRAG is *addwords* (figure 6) that adds low and high words of a 32-bit input during checksum computation.

6 Results and Analysis

In this section we present some results of our instruction set specialization and analyze them in detail. Ta-

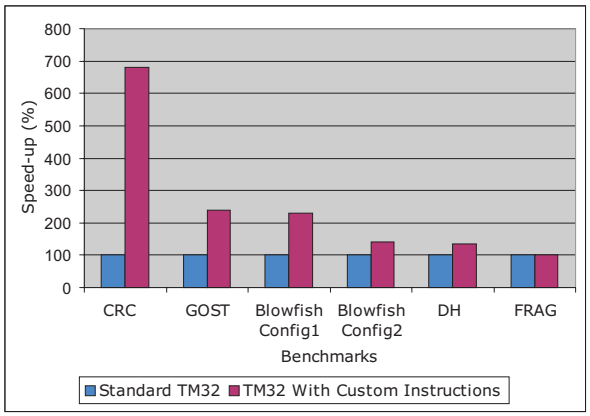

Figure 7. Relative Speed-ups for Different Applications

ble 2 presents the estimated latency and area overhead for different custom instructions. Figure 7 shows the speed-ups obtained by implementing different custom instructions in the TriMedia model. Figure 8 shows the area overhead for different custom instructions w. r. t. to a single multiplication operation (since multiplication is the most costly operation in our current model). For Blowfish, we have evaluated two configurations of custom instructions. *Configuration 1* consists of only the *bff*, *configuration 2* consists of three instructions - *saccess1, saccess2* and *addxoradd*[6].

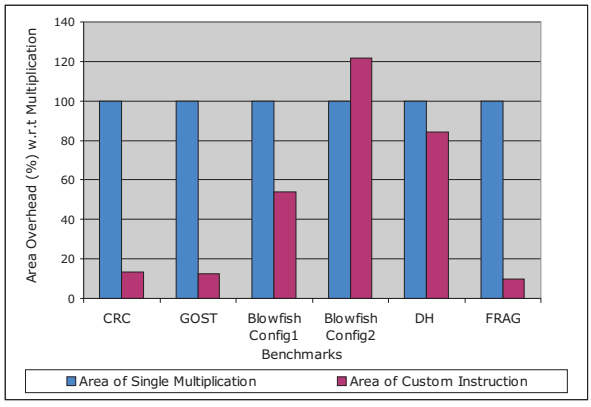

Figure 8. Area Overhead of Custom Instructions w.r.t. Multiplication

The latency and area overhead values have been obtained using the method described in section 4. For custom instructions with internal memory accesses (i.e. *saccess1, saccess2,* and GOST and Blowfish F functions), the area and latency estimation only takes into account the combinational logic (including the memory access hardware) and completely ignores the memory access latencies. The rationale behind this assumption is described separately in subsection 6.1.

[6]The area overhead takes into account the area for an entire configuration. For example, consider the second configuration for Blowfish. To make this configuration effective, the instruction saccess2 must be implemented in 3 parallel slots. Therefore, the total area for this configuration is (3 * area of saccess2 + area of saccess1 + area of addxoradd)

Table 2. Area and Latency of Custom Instructions

Benchmark	Instruction	Latency	Area w.r.t. Mult(%)
CRC	reflect8	1	6.7
	reflect32	1	6.8
Blowfish (Config. 1)	bff	2	53.94
Blowfish (Config. 2)	addxoradd	2	29.2
	saccess1	2	24.9
	saccess2	2	22.6
GOST	gostf	2	12.6
DH	cimul	4	84.3
FRAG	addwords	2	10

As can be easily seen, for applications with well defined *hot-spots* like Blowfish, GOST and CRC, custom instructions can produce significant speed-ups at a very low area overhead. For these three applications, we obtained an average speed-up (considering the best configuration for Blowfish) of around 380% with a total area overhead of around 80% w. r. t. a single multiplication unit. Considering that TM32 has a total of 28 functional units (including division and floating-point units), this area overhead is very insignificant. Also, observe that an application can not benefit from special instructions, if it does not have a well defined hot-spot (like Diffie-Hellman and FRAG in our case). Therefore, it is not wise to use instruction set customization for such applications.

It is to be noted that due to the diversity of the benchmarks selected, the identified custom instructions vary widely from benchmark to benchmark. This indicates that a *generic* customized instruction set for the entire network processing area is almost impossible to find. Therefore, the functionality of each network layer must be investigated separately for instruction set customization. At the same time, there exist a number of common functions, such as CRC, that can be found in different layers. Such functions can immensely benefit from special instruction sets.

6.1 Latency Estimation for GOST and Blowfish Custom Instructions

The instructions with internal memory accesses from GOST and Blowfish deserve our special attention. Such instructions disturb the RISC philosophy of the target architecture and increases the number of memory ports with an overall increase in area and power consumption. Moreover, their low latencies, given the number of internal parallel memory accesses, might seem unrealistic. However, in designing these custom instructions we took the advantage of a peculiarity of private-key block cipher algorithms and recent advancements of memory-hierarchy design.

Almost all private key block-ciphers, such as AES, DES, Blowfish, GOST etc, uses *S-Box* substitution for encryption. These *S-Box*es are constants that are either defined by the standard or is generated as part of the encryption process. Throughout the encryption process, they are accessed most heavily. Recent researches [4][3] suggest to keep such heavily accessed

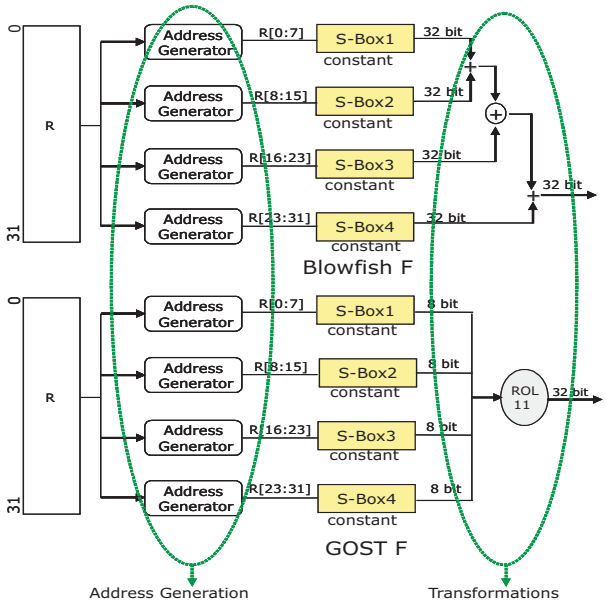

Figure 9. Common Characteristics of Blowfish and GOST *F* Functions

data objects in *local scratch-pad* memories. Scratch-pad memories are very fast, expensive memories similar to on-chip caches. However, while the placement and replacement of data objects in caches is decided dynamically during execution, scratch-pads are fully compiler controlled. The compiler can place frequently accessed data objects in scratch-pads eliminating the access latencies completely. [7] The *S-Box*es in GOST and Blowfish are good candidates to be kept inside such scratch-pads. With this small architectural modification, it is possible to meet the latency constraints for both GOST and Blowfish custom instructions.

In fact, the symmetric-key block cipher algorithms are so closely related to each other that a generic custom instruction can be designed for them. In figure 9, the *F* functions for Blowfish and GOST are shown side by side. As can be easily seen, both the functions are extremely similar - first they apply some *address generation* technique on the input data to access the *S-Boxes*, then they *transform and combine* the *S-Box* outputs. Such similar behavior can be easily combined in a *generic F* function. By putting the *S-Boxes* into local memory the memory latencies can be fully eliminated. This can result in significant improvements of performance and power consumption.

7 Summary and Outlook

In this paper, we have presented a case study of *Instruction Set Specialization* for network processing ASIPs. Using TriMedia VLIW Processor as the basic template and using state-of-the-art ADL based architecture exploration tools, we have evaluated several custom instructions for different network processing applications. Our studies reveal some promising custom instructions in the areas of private-key cryptography and error detection that can speed-up application execution to a large extent. Additionally, our results suggest that a generic custom instruction set may not exist for network processing as a whole.

In future, we plan to extend our work to take into account the effects of different memory architectures. In this study, we have always assumed that heavily accessed data objects are placed inside fast, local memories. The advantage of this approach in terms of performance and power consumption must be properly studied. Another unexplored area is the integration of the custom instructions into a compiler tool-chain which will be addressed in future.

[7] Since scratch-pad memories are expensive, they are small in size and only a few selected, heavily accessed data objects can be placed inside them. They are not very useful when an application has a uniform access pattern to a large number of data objects.

References

[1] F. Sun, S. Ravi, A. Ragnunathan, N.K. Jha: *Synthesis of Custom Processors based on Extensible Platforms*, ICCAD, 2002

[2] D. Goodwin, D. Petkov: *Automatic Generation of Application Specific Processors*, CASES, 2003

[3] P. Biswas, V. Choudhary, K. Atasu, L. Pozzi, P. Ienne, N. Dutt: *Introduction of Local Memory Elements in Instruction Set Extensions*, Proceedings of the Design Automation Conference, June 2004

[4] S. Steinke, L. Wehmeyer, B.S. Lee, P. Marwadel: *Assigning Program and Data Objects to Scratch-pad for Energy Reduction*, University of Dortmund

[5] H.Scharwaechter, D. Kammler, A. Wieferink,M. Hohenauer, J. Zeng, K. Karuri,R. Leupers,G. Ascheid, H. Meyr:*ASIP Architecture Exploration for Efficient IPSec Encryption: A Case Study*, 8th International Workshop on Software & Compilers for Embedded Systems (SCOPES), Sept. 2004

[6] H. Xie, L. Zhao, L. Bhuyan: *Architectural Analysis of Cryptographic Applications for Network Processors*,Proceedings of the IEEE First Workshop on Network Processors with HPCA-8, February 2002

[7] J. Burke, J. McDonald, T. M. Austin: *Architectural Support for Fast Symmetric-Key Cryptography*,Architectural Support for Programming Languages and Operating Systems, pp.178-189, 2000

[8] K. Atasu, L. Pozzi, P. Ienne:*Automatic Application-Specific Instruction-Set Extensions under Micro-architectural Constraints*, Proceedings of the 40th Design Automation Conference, June 2003

[9] N. Shah,Understanding Network Processors Ver 1.0,2001

[10] I. A. Zabotin, G. P. Glazkov, V. B. Isaeva : *Cryptographic Protection for Information Processing Systems: Cryptographic Transformation Algorithm, publisher : Government Standard of the USSR* , 1989

[11] W. Diffie, M. E. Hellman: *New Directions in Cryptography*,IEEE Transactions on Information Theory, vol.17, pp.46-54, July2003

[12] Philips Semiconductors, http://www.semiconductors.philips.com/news/content/file_234.html

[13] J. Allen,B. Bass,C. Basso, R. Boivie, J. Calvignac, G. Davis, L. Freléchoux, M. Heddes, A. Herkersdorf, A. Kind,J. Logan, M. Peyravian, M. Rinaldi, R. Sabhikhi, M. Siegel , *IBM PowerNP Network Processor: Hardware Software and Applications*, IBM Journal of Research and Development,Vol.47,2003,p 177-194

[14] Intel IXP12XX Product Line of Network Processors http://www.intel.com/design/network/products/npfamily/ixp1200.htm

[15] Parallel Express Forwarding on the Cisco 10000 Series http://www.cisco.com/warp/public/cc/pd/rt/10000/prodlit/pxfw_wp.pdf

[16] CommBench, http://ccrc.wustl.edu/ wolf/cb/

[17] NetBench, http://cares.icsl.ucla.edu/NetBench/

[18] CRC Implementation by Sven Reifegerste, http://rcswww.urz.tu-dresden.de/ sr21/crctester.c

[19] LISATek products, http://www.coware.com

[20] TriMedia32 Architecture,Preliminary Specification,2003

[21] RFC 793: Transmission Control Protocol, http://www.faqs.org/rfcs/rfc793.html

[22] A. S. Tanenbaum,*Computer Networks*, Prentice-Hall Pvt Ltd,ISBN 81-203-0621-X,

[23] B. Schneier,*Applied Cryptography Second Edition: Protocols, Algorithms and Source Code in C*,John Wiley & Sons, Inc.,January 1996

[24] Synopsys Design Compiler Overview, http://www.synopsys.com/products/logic/design_comp_cs.html