TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 03/21 pp19-28
Volume 14, Number 1, February 2009

Architecture-Aware Session Lookup Design for Inline Deep Inspection on Network Processors*

XU Bo (徐 波)[1,2], HE Fei (何 飞)[1,2], XUE Yibo (薛一波)[2,3], LI Jun (李 军)[2,3,**]

1. Department of Automation, Tsinghua University, Beijing 100084, China;
2. Research Institute of Information Technology (RIIT), Tsinghua University, Beijing 100084, China;
3. Tsinghua National Laboratory for Information Science and Technology (TNList), Beijing 100084, China

Abstract: Today's firewalls and security gateways are required to not only block unauthorized accesses by authenticating packet headers, but also inspect flow payloads against malicious intrusions. Deep inspection emerges as a seamless integration of packet classification for access control and pattern matching for intrusion prevention. The two function blocks are linked together via well-designed session lookup schemes. This paper presents an architecture-aware session lookup scheme for deep inspection on network processors (NPs). Test results show that the proposed session data structure and integration approach can achieve the OC-48 line rate (2.5 Gbps) with inline stateful content inspection on the Intel IXP2850 NP. This work provides an insight into application design and implementation on NPs and principles for performance tuning of NP-based programming such as data allocation, task partitioning, latency hiding, and thread synchronization.

Key words: session lookup; deep inspection; network processor; performance optimization

Introduction

Traditional firewalls are designed to protect local networks from unauthorized access according to access control lists (ACLs). However, with the emergence of stateful inspection in recent decades, various implementations of stateful session maintenance strategies have been developed to achieve high-speed session creation, lookup, and teardown inside the security appliances. Nowadays, security gateways are taking charge of not only blocking malicious attackers by verifying packet headers, but also scanning flow payloads against deliberate intrusions. These requirements have stimulated the research on deep inspection, which seamlessly integrates packet classification for access control and pattern matching for intrusion prevention through a session lookup strategy.

With the rapid increase of network bandwidth, general purpose processors (GPPs) are becoming more and more incompetent to catch up with the performance requirement for the categorization of incoming and outgoing packets into corresponding network flows at the OC-48 speed or higher. In addition, with the ever-changing network environments and the newly-emerging types of attacks, the long cycle and high cost of ASIC research and development makes it infeasible to meet the time-to-market demands of today's network appliances. Consequently, network processors (NPs) are becoming extremely attractive alternatives in high-end security gateway design.

NPs are anticipated to provide the same high

Received: 2008-03-26; revised: 2008-10-14
* Supported by the Basic Research Foundation of Tsinghua National Laboratory for Information Science and Technology (TNList) and the National High-Tech Research and Development (863) Program of China (No. 2007AA01Z468)
** To whom correspondence should be addressed.
E-mail: junl@tsinghua.edu.cn; Tel: 86-10-62796400

performance as ASICs and the time-to-market advantage of GPPs. The main characteristics of NPs are the highly optimized hardware architecture for high speed network computing and packet processing, the distributed, multiprocessor, multithreaded architectures, and the programming flexibility. In recent years, many integrated circuit companies, such as Intel[1], AMCC[2], Freescale[3], and Agere[4], have developed their own programmable NPs. Cavium[5] and RMI[6] have also paid much attention on NPs with their relatively new multiple MIPS core solutions.

This paper focuses on NP-based high performance inline stateful deep inspection, which is the dominant function block in intrusion prevention systems (IPSs). This requires well-designed session data structures along with efficient implementation on NPs. The main challenges of this research include:

(1) **Secure stateful processing** The session table should support the security mechanisms to guarantee secure stateful processing. TCP validations including three-way handshake flag checking and sequence number and acknowledgement number checking should be used to verify the legitimacy of packets and flows to eliminate illegal packets or flows. Moreover, the session table should be designed with consideration of out-of-order packets and reserve memory for unordered packets.

(2) **Flow-level deep inspection** Today's network intrusions are more difficult to detect due to the ever-complicated attack modes spanning across packets. The evasion attack is quite dangerous since its signature is not contained in a single packet but divided into several segments and placed in two or more consecutive packets of the same flow. The signature is reconstructed when the packets reach the victim. Thus, interim security devices cannot catch it by packet-level intrusion detection. The only way to detect evasion attacks is flow-level deep inspection, which scans malicious signatures against the flow content rather than the payload of individual packets.

(3) **Efficient implementation on NPs** Although NPs provide an excellent candidate for network processing, programming on multi-core, multithreaded NPs is a big challenge because experience with general purpose multi-processing programming is not directly applicable to such system-on-chip (SoC) architectures. For example, latency hiding is an important issue when programming on micro-engines (MEs), and mutual exclusion operations must be considered when more than two MEs are applying to access the same memory address. In addition, the implementation should utilize the characteristics of the NP platform to achieve high performance.

This work investigates architecture-aware session design on NPs, and provides an instance of implementation to exploit the parallelism of the multi-core, multithreaded NP. The main contributions of this paper include:

(1) **Architecture-aware session data structure design** A well-designed session data structure is proposed to support TCP validation and inline stateful deep inspection. The scheme provides fast-path implementation of stateful session lookup and flow-level deep inspection, as well as slow-path unordered packet buffering and TCP reassembly. Moreover, the design takes advantage of the NP characteristics to achieve an inline deep inspection speed of 2.5 Gbps.

(2) **Performance optimization on NPs** This paper gives an example of how to efficiently implement applications on NPs and investigates the main programming issues: memory space reduction, data allocation, task partitioning, latency hiding, and thread synchronization, which have drastic performance impacts on NP-based application implementations.

(3) **Mechanism for integrating inline content inspection** By buffering unordered packets in flows and reassembling them, flow-level inline content inspection is realized based on single packet payload inspection. According to a previous observation, the proportion of unordered flows is less than 3%[7], so there are only a few flows that need buffers to store unordered packets. Besides, each flow engages a buffer caching the last characters of the previous packet to combine the flow contents. The buffer size is typically determined by the length of the longest signature. Consequently, the flow-level content inspection can achieve an inline processing speed of 2.5 Gbps with modest memory storage.

1 Related Work

Researchers have been trying hard to improve the performance of deep inspection based on FPGAs/ASICs. Some have achieved the OC-48 rate for intrusion detection, but not on an NP platform.

Schuehler and Lockwood[8] took advantage of FPGA to implement stateful flow tracking, TCP stream reassembly, context storage, and flow manipulation services in 2004. They designed a circuit capable of monitoring bidirectional TCP flows at OC-48 data rate. A TCP/IP flow monitoring system called TCP-splitter was also implemented based on reconfigurable hardware for analyzing and processing TCP/IP flows at OC-48 line rate[9].

Dharmapurikar et al.[10] developed a technique based on Bloom filters to detect predefined signatures in the packet payload. With the state-of-the-art FPGAs, the scheme can support 10 000 strings at the OC-48 line rate. Moscola et al.[11] designed another content-scanning module for an Internet firewall using finite state machines (FSMs), and it was also implemented on FPGA and reached the OC-48 line speed.

However, the FPGA-based solutions suffer badly from hardware inflexibility. Any change in data structures or signatures requires recompilation, regeneration, and replacement of the circuits on FPGA platforms. In contrast, the powerful computing ability and the high flexibility of NPs leverage the advantages of GPP and FPGA. If the architecture characteristics can be effectively exploited, a high performance inline stateful deep inspection system could be realized on NPs.

Some researchers have focused on NP-based firewall design and implementation. Shen et al.[12] proposed an optimized firewall design based on the Intel IXP 2400 NP. They designed a framework for stateful firewall systems, and optimized the access control list and status session table creation and lookup. The firewall system achieves the OC-48 line speed using hardware implementations.

Zhong et al.[13] presented a comprehensive IPv4/IPv6 firewall system based on NPs. They introduced the firmware and software structures of the synthetic firewall system, invoking the multithread characteristics of the Intel IXP 2400 NP. The system is believed to be adaptable to the next generation Internet.

Although these previous works have achieved the OC-48 line speed, they are at most stateful firewall systems rather than deep inspection systems with inline stateful content inspection. The demand for high performance deep inspection greatly motivates this research.

2 Architecture-Aware Session Design

The session design aims to achieve deep inspection on NPs, with packet classification for access control and pattern matching for intrusion prevention into a seamless integration. This objective poses two main challenges: (1) the integrated system has a tighter performance budget for each single packet to traverse through the session table; (2) the integration with inline content inspection raises the demand for session entries to store the out-of-order flag, cache the unordered packets, and store the current characters of each flow. Hence, the session data structure is of great importance to inline deep inspection, and its design should take advantage of the parallelism of the multi-core, multithreaded NPs, while consider the mutual exclusion of memory accesses in different MEs. Session data structure includes two components: session entry data structure and session table data structure.

2.1 Session entry data structure

The session entry data structure is the data structure of each session entry. The session entry data structure designed in this paper is illustrated in Table 1. The total structure size is 72 bytes. LW is short for long word, with each long word 4 bytes in length. The session entry data structure can be segmented into three parts.

Part 1 includes LW0 to LW3 where the higher 16 bits of LW0 indicate the next session pointer of the session entry, while the lower 16 bits of LW0 are used to store the IP protocol, classification policy, IDS flag, and the mutual exclusion (mutex) lock for avoiding access collisions between threads of the MEs. Once a thread accesses the session entry data structure, the mutex lock is set true so that no other thread can access the same session entry. Concurrent read operations by different threads do not need to be excluded since they do not change the memory data. LW1 to LW3 store the four-tuple header information of the flow.

Part 2 from LW4 to LW13 maintains all the necessary information for the upstream and downstream flows of the same connection. The upstream represents the flow from the originating client to the replying server while the downstream represents the flow in the opposite direction. The upstream or downstream connection information includes the sequence number, the

Table 1 Session data structure

LW	Bits	Size	Field description
0	31:16	16	Next session pointer
0	15:15	1	Mutex Lock
0	14:12	3	IDS flag
0	11:8	4	Classification policy
0	7:0	8	Protocol
1	31:0	32	Source IP address
2	31:0	32	Destination IP address
3	31:16	16	Source port
3	15:0	16	Destination port
4	31:0	32	Upstream Seq No.
5	31:0	32	Upstream Ack No.
6	31:16	16	Upstream code bits
6	15:0	16	Upstream win size
7	31:0	32	Upstream time stamp
8	31:0	32	Upstream session length
9	31:0	32	Downstream Seq No.
10	31:0	32	Downstream Ack No.
11	31:16	32	Downstream code bits
11	15:0	32	Downstream win size
12	31:0	32	Downstream time stamp
13	31:0	32	Downstream session length
14	31:31	1	Upstream out-of-order flag
14	30:24	7	Upstream out-of-order No.
14	23:0	24	Upstream buffer head pointer
15	31:0	32	Upstream joint buffer address
16	31:31	1	Downstream out-of-order flag
16	30:24	7	Downstream out-of-order No.
16	23:0	24	Downstream buffer head pointer
17	31:0	32	Downstream joint buffer address

acknowledge sequence number, the TCP code bits, the windows size, the time stamp, and the session traffic length. These fields are reserved to do TCP validation and check the packet ordering.

Part 3 from LW14 to LW17 enables the session to deal with out-of-order flow packets as well as inline stateful content inspection. Bit 31 of LW14 indicates whether there are unordered packets in upstream flow. Bits 30-24 count the unordered packet number while bits 23-0 store the header pointer of the buffer that caches the out-of-order packets. Based on prior statistics[7], less than 5% of the out-of-order flows have more than one hole. Therefore, we only cache the flows with one hole and set a threshold of unordered packets number in one flow. LW15 indicates the address of the joint buffer which stores the last characters of the previous packet. These characters are combined with the coming packet payload for flow-level content inspection. The joint buffer is placed in scratchpad to accelerate access. The size of the joint buffer should be adjusted according to the longest length of signatures. LW16 to LW17 indicate the out-of-order buffer address and the joint buffer address for the downstream flow.

The dedicated session entry data structure design facilitates inline deep inspection based on stateful packet processing and single packet payload inspection.

2.2 Session table data structure

The session table data structure defines how the session entries are organized. The conventional hash table is modified to allow dynamic link lists in our implementation as shown in Fig. 1. Furthermore, a memory

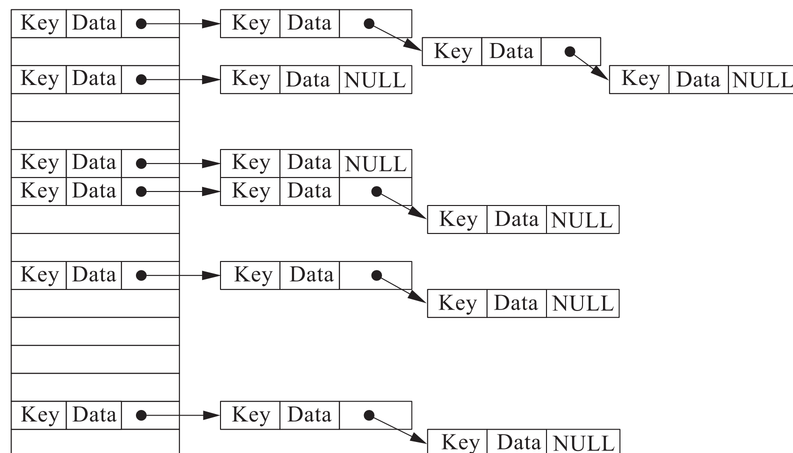

Fig. 1 Hash table data structure

ring is designed in this paper, as shown in Fig. 2, to facilitate the insertion and deletion of the dynamic session entries.

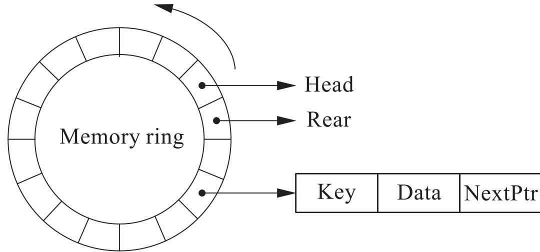

Fig. 2 Memory ring for dynamic allocation

As shown in Fig. 1, the left column of the hash table is an array called the fixed session table, which stores the header entry of each hash link list. The mutex lock in this session entry is the access token for the whole link list. The session entries linked behind the pointers are dynamically allocated when there are collisions. A memory block-named the dynamic session table is reserved for these dynamic allocations and repeals and an indicator-named the session allocation flag is used to maintain the allocation and release of the dynamic session entries.

Assuming a load factor of 1/2, with 100K (K=1024) simultaneous sessions at the same time, a total of 200K session entries are required for the fixed session table. In addition, 100K session entries are reserved for dynamic entry allocations. The load factor α is defined by n/m, where m is the total number of buckets in the hash table and n is the maximum number of buckets that are simultaneously occupied. As a result, the total memory used for the hash table will be $(200\text{K}+100\text{K})\times 72\times 8\text{ bits}=172.8\text{ Mbit}$, which can be stored in SRAM on the Intel IXP2850 NP.

3 Implementation on Network Processors

Network processors, beneficial from their programmable architectures, are emerging as attractive candidates for network processing and will gradually be widely used for applications in network appliances such as switches, firewalls, and security gateways. Being highly optimized for fast network computing and packet processing, NPs are capable of hiding memory access latencies to attain high processing rates typically by the distributed, multiprocessor, multithreaded architectures. In this paper, the Intel IXP2850 NP is chosen as the implementation platform.

3.1 Intel IXP2850 NP and processing stages

Figure 3 illustrates the components of the Intel IXP2850 NP, which include 1 XScale core, 16 MEs, 4 SRAM controllers, 3 DRAM controllers, and high-speed bus interfaces. The XScale core is a general purpose 32-bit RISC processor, which initializes and manages the MEs and handles higher layer network processing tasks. Each ME has eight hardware-assisted execution threads and 640 words single-cycle access local memory. There is no cache on the MEs. Each ME uses the shared buses to access off-chip SRAM and

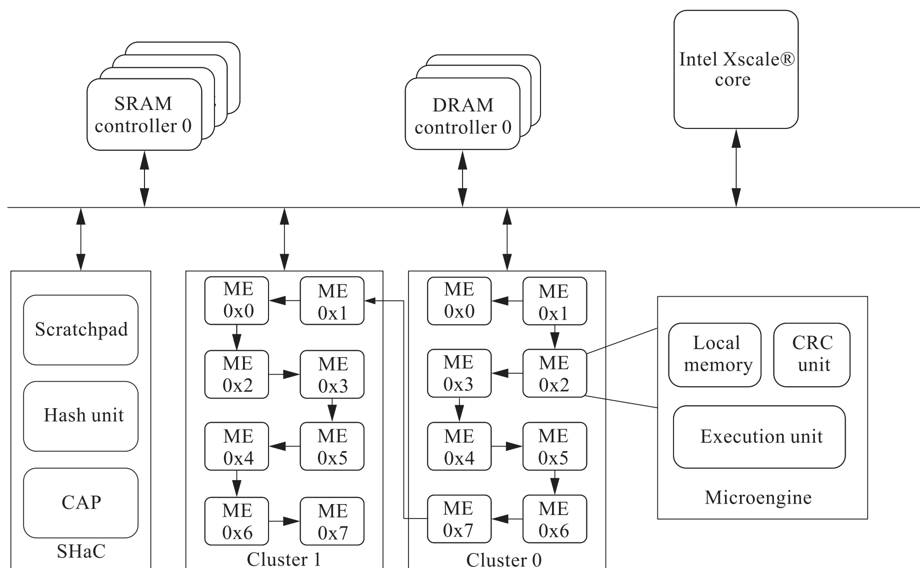

Fig. 3 Intel IXP2850 hardware blocks

DRAM banks. The average access latency for SRAM is about 150 cycles and that for DRAM is about 300 cycles. The session prototype is implemented with inline content inspection with assistance of the IXA SDK4.0 Workbench and simulated on a cycle-accurate simulator. In the following, the packet processing stages and hardware design issues in the implementation of the session prototype are discussed for achieving the high speed of OC-48.

The session prototype utilizes six packet processing stages (PPSs). First, the packet receive PPS receives the Ethernet packets from the MSF RBUF and reassembles the RBUF data into one or more packet buffers. Then, the session lookup PPS with the inline content inspection is evoked along with a bypassing session update PPS to update corresponding fields in the session data structure. After that, the IPv4 packets are forwarded to the packet scheduling PPS to protect the order and finally the packets are transmitted through the CSIX fabric with a queue manager PPS that enqueues and dequeues the packets. The processing stages are shown in Fig. 4 and the pseudo-code for the session processing PPS is shown in Fig. 5.

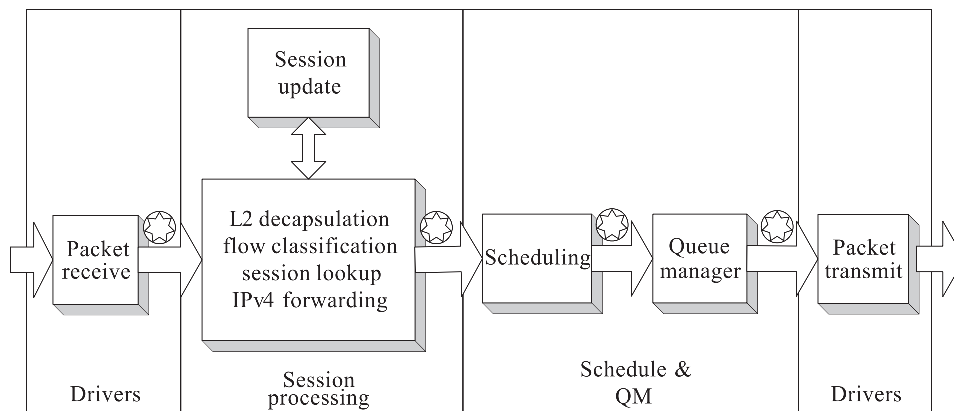

Fig. 4 NP processing stages

```
PPS Session_Lookup_Pps(void)
{
    for (;;)
    {
    dlNextBlock = Dl_Source();
    dlNextBlock == Ether_Decap();
    /* Session Lookup */
    hashKey = Hash (ipv4TcpHdr);
    while (1)
    {
       If (0 == atomic_test_and_set(mutex_lock) )
       {
       session_exist = Check_Session_LinkList_Exist(hashKey);
       if (! Session_exist)
            Session_Create(currentSessionPtr, ipv4TcpHdr);
       else
       {
            while (1)
            {
               If (Match_Ipv4TcpHdr(currentSessionPtr,
                        ipv4TcpHdr))
                 Session_Update(currentSessionPtr, ipv4TcpHdr);
               else
               {
                 currentSessionPtr->nextSessionPtr =
                          Session_Alloc();
                 Session_Create (currentSessionPtr->nextSessionPtr,
                          Ipv4TcpHdr);
               }
            }
       }
       atomic_clear (mutex_lock);
       }
    }
    dlNextBlock == Ipv4_Forward();
    Dl_Sink();
    }
}
```

Fig. 5 Session processing PPS pseudo-code

3.2 NP-based design principles

Since the session prototype is implemented on a multi-core, multithreaded NP, the programming needs to be architecture-aware to take advantage of the hardware characteristics. For instance, to hide memory access latencies, fine-grained interactions are used to connect tasks on the same ME. Furthermore, the NP needs to process packets in real time, which makes the performance budget for single packet processing rather tight. To achieve the OC-48 line rate, at most 228 ME clock cycles are allowed on the Intel IXP2850 NP to process a minimal IPv4 packet. With such a rigid restriction, the data-path programming on the multi-core, multithreaded NPs must consider the following aspects:

(1) Primary parallel programming issues, such as data allocation and task partitioning, to minimize memory access latencies;

(2) Hardware architecture specific factors that have crucial impact on performance, such as instruction selection and intrinsic function invoking;

(3) Thread-level parallelism for hiding memory access latencies;

(4) Thread synchronization and mutual exclusion for coordinating potential parallelism between threads;

(5) Limitations of on-chip local memory or control store for the code size, and the register numbers allocated for each thread.

The benefits from these optimization strategies may vary from a few cycles to hundreds of cycles. For example, reallocating data from DRAM to SRAM on the Intel IXP2850 can save nearly 150 clock cycles per memory access. Besides, if two memory accesses are scheduled in consecutive cycles, the issuing cycles of the second memory access can be completely hidden. Through deliberate application design and appropriate hardware mapping from high-level decisions on data allocation and task partitioning down to low-level decisions such as instruction selection and scheduling, high performance can be achieved on NPs. The extraordinary hardware characteristics of the NPs must be excavated to obtain the performance gain. Detailed experiments of the optimization techniques will be discussed in Section 4.2.

4 Simulation and Performance Analysis

To evaluate the performance of the session prototype proposed in this paper, we set a series of experiments that mainly focused on the session lookup performance with inline stateful content inspection and the performance impact of the architecture-aware optimization decisions, which provide insight into programming on the multi-core, multithreaded NPs.

4.1 Session lookup performance evaluation

Since the objective is to achieve high performance session lookup with inline stateful content inspection, the performance was evaluated on three aspects: (1) session creation speed that reflects the ability of the session prototype to meet the requests for creating new concurrent sessions; (2) session lookup speed without content inspection that reflects the performance of flow classification; and (3) session lookup speed with inline stateful content inspection, which provides the overall performance of the session prototype with deep inspection enabled.

4.1.1 Experiment setup

The flow traces were real-life packets collected at the edge firewall of the Research Institute of Information Technology, Tsinghua University. The four-tuple packet header information was used to generate hash keys with CRC Hash selected for this implementation. Thus, with a load factor of 1/2, the hash collision rate is 10.68% in the tests. Moreover, if 100K concurrent sessions are to be supported, 200K session entries should be reserved in the fixed session table and another 100K session entries should be reserved in the dynamic session table, organized in memory rings. Thus, a total of 172.8M (M=1024^2) memory is needed.

The hash table data structure is allocated in SRAM with the simulation platform being the Intel IXP2850 NP with the IXA SDK4.0 Workbench.

The session update PPS is responsible for eliminating out-of-date sessions, which ensures the renewing of the session entries. Referring to some industrial products, the session timeout interval was set to 30 s.

The session lookup speed with inline stateful content inspection was evaluated using the recursive shift indexing algorithm[14] as the pattern matching algorithm in the session lookup PPS, due to its higher matching speed compared with other pattern string matching algorithms. The 2835 intrusion signatures used in the tests are seized from Snort 2.6[15], the well-known open source IDS. Besides, the red traffic and orange traffic archives on the Snort website were used to generate the packet payloads for the flows. Thus, the tests simulate real-life circumstances, with real signatures in the traffic payload. Moreover, to achieve stateful content inspection, joint buffers are hired in session table for each live flow. The joint buffers were allocated in scratchpad in our experiments, in order to accelerate the memory accesses at the buffer content switches. Generally, one scratchpad access needs 60 clock cycles while the average access latency for SRAM is 150 cycles and for DRAM is 300 cycles.

4.1.2 Experimental results

Table 2 shows the memory requirements for the session data structure, which show that the memory occupation is proportional to the session size m, i.e., the number of buckets. If the number of supported concurrent sessions is k, then for the load factor of 1/2, m equals $2k$. In our implementation, m entries were reserved in the fixed session table and $m/2$ entries were

reserved in the dynamic session table. The total memory size was ($2k + k$) multiplied by the size of one session entry, which is 72 bytes in this design.

Table 2 Memory occupation of session table structure

Concurrent session number	Memory (KB)
10 000	17 280
100 000	172 800
500 000	864 000
1 000 000	1 728 000

Figure 6 shows the session creation rate for the session prototype. Aiming at being placed at edge networks as security gateways, the prototype is designed to support a large number of concurrent sessions. With real-life flow packets, the flow rate was about 14.5M connections per second (14.5 Mcps) with the load factor of 1/2. Moreover, the time performance scales well as the number of concurrent sessions increases, indicating that the hash function time performance is mostly influenced by the load factor.

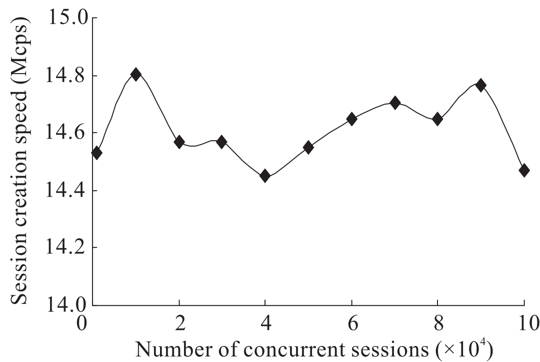

Fig. 6 Session creation performance

Figure 7 compares the session lookup performance with and without inline content inspection. The prototype achieves a lookup speed of 7.5 Gbps without content inspection with all the TCP states tracked and verified in the session data structure. A simple version of Layer 3 IP validation and Layer 4 TCP validation is synthesized in the implementation, including payload length checking, sequence/acknowledgement number checking, and TCP code bits checking. It is believed a complete protocol validation via stateful trace tracking can be performed based on this platform. Besides, the session lookup time scalability indicates that the system can support more concurrent sessions.

Figure 7 also shows the session lookup rate with inline stateful content inspection. The recursive shift

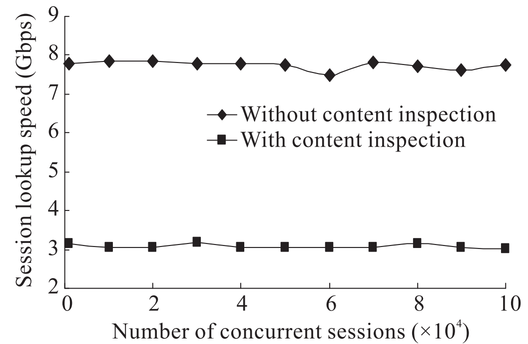

Fig. 7 Session lookup performance

indexing algorithm reduces unnecessary comparisons via recursive shifting. It shows that when embedded in the session lookup stage, the algorithm reaches a processing speed higher than the OC-48 line rate, which is an appealing result on NPs.

4.2 Architecture-aware design issues

4.2.1 Data allocation

Like many other NPs, the Intel IXP2850 has a rich memory hierarchy including local memory, scratchpad, SRAM, and DRAM. For the session prototype, the allocation of the session table structure greatly affects the session creation and lookup speeds. Meanwhile, for the content inspection algorithm, the allocation of the preprocessing data structures also drastically influences the signature matching speed.

To evaluate the memory distribution impacts, we tested four data allocation schemes on the session lookup performance with inline content inspection. The session table size was set to 100K entries and experimental results are shown in Table 3.

Table 3 Lookup rate on various data allocations

	SRAM	Hybrid-1	Hybrid-2	DRAM
2 MEs	1.97G	1.86G	1.23G	1.14G
4 MEs	3.90G	3.75G	2.47G	2.35G
8 MEs	7.85G	7.58G	4.79G	4.63G

Note: $G=1024^3$

In Table 3, the SRAM scheme places both the fixed session table and the dynamic session table in SRAM while the DRAM scheme places both tables in DRAM. The Hybrid-1 scheme places the fixed session table in SRAM and the dynamic session table in DRAM, while the Hybrid-2 scheme places the fixed session table in DRAM and the dynamic session table in SRAM. The results show that placing data in SRAM gives better

performance than in DRAM. Thus, NP programs should allocate data in faster memory banks as much as possible. However, the faster memory banks are always smaller in size so programmers must balance the memory allocation verses the overall performance.

4.2.2 Task partitioning

The IXP2850 NP has 16 MEs in total, which raises a question as to how to distribute the computation power. Generally, the computing resources of MEs should be divided according to two principles: (1) the different functions of the microblocks; and (2) the performance budget and overhead of the microblocks. The session prototype has six PPSs. The 16 MEs are distributed as follows: 2 for the packet receive PPS, 8 for the session lookup PPS, 2 for the session update PPS, 1 for the packet scheduling PPS, 1 for the queue manager PPS, and 2 for the CSIX transmit PPS.

Tests showed that the session lookup PPS is the bottleneck that intensively limits the overall throughput. The results for various task partitioning strategies including multi-processing and context-pipelining are shown in Table 4. The context-pipelining is not suitable for the session prototype due to the dynamic nature of the workload.

Table 4 Lookup rate under different task partitioning schemes

	Multi-processing	Context-pipelining
2 MEs	1.97G	—
4 MEs	3.90G	—
6 MEs	5.93G	2.11G
8 MEs	7.85G	2.01G

4.2.3 Latency hiding

Hiding memory latencies is the key issue in achieving high performance applications. The memory access latencies are hidden typically by overlapping the memory access with the arithmetic computations in the same thread. The microengineC compiler provides a switch to turn latency hiding optimizations on or off. Thus, the compiler can schedule ALU instructions into the delay slots of a conditional branch instruction and a memory access instruction. When using microcode, programmers should be conscious of hiding memory latencies, though the assembler also has its own optimization options. In Fig. 5, line 8 is for calculating the hash key while line 11 is for writing the SRAM to set the mutex lock. These two instructions can run in parallel so that the hash key computation is hidden completely by the memory write operation. There are more instances in the real code.

4.2.4 Thread synchronization

Thread synchronization is a typical problem that programmers face on NPs due to their multi-core and multithreaded architectures. Different threads on the same ME or different MEs may compete for the sharing resources such as SRAM channels, or even the same SRAM entry addresses when there are collisions in the hash function. Therefore, atomic read and write operations on SRAM are engaged in this implementation. The mutex locking is needed for ensuring the logical correctness of the application. However, it brings a 6%-10% overhead according to the test results listed in Table 5. The session lookup without inline content inspection is taken in this test.

Table 5 Thread synchronization overhead

	With mutex	Without mutex	Overhead (%)
2 MEs	1.97G	2.13G	8.1
4 MEs	3.90G	4.30G	10.3
6 MEs	5.93G	6.42G	8.3
8 MEs	7.85G	8.37G	6.6

5 Conclusions and Future Work

Current firewalls and security gateways not only block unauthorized accesses by inspecting packet headers, but also inspect flow contents against malicious intrusions, which significantly motivate the research on deep inspection. Deep inspection aims at combining packet classification for access control and pattern matching for intrusion prevention into a seamless integration. This paper proposed a well-designed session lookup scheme for inline stateful content inspection and implemented it efficiently on the IXP2850 NP platform. With the dedicated session data structure and integration approach, the OC-48 line rate stateful inline content inspection is achieved. By means of excavating the principles on NP-based performance tuning such as data allocation, task partitioning, latency hiding, and thread synchronization, we realized an architecture-aware session design to promote the performance of the integration system and provide an insight into application design and implementation on NP. Future work will be conducted to implement high level deep inspection such as Layer 7 protocol analysis and Web

content filtering. Moreover, security mechanisms such as TCP proxy and anti-DoS are within our future research interests.

Acknowledgements

The authors would like to acknowledge Qi Yaxuan, Yu Jianming, and Zhou Xin for their suggestions and help. Thanks also to all of our other colleagues in the Network Security Lab for their support and advice.

References

[1] Intel, IXP2XXX Product Line of Network Processor, http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm, 2008.

[2] AMCC, Network Processor, https://www.amcc.com/MyAMCC/jsp/public/browse/controller.jsp?networkLevel=COMM&superFamily=NETP, 2008.

[3] Freescale, C-Port Network Processors, http://www.freescale.com/webapp/sps/site/homepage.jsp?nodeId=02VS01DFTQ3126, 2008.

[4] Agere, Network Processor, http://www.agere.com/telecom/network_processors.html, 2008.

[5] Cavium, http://www.caviurm.com/, 2008.

[6] RMI, http://www.razamicroelectronics.com/, 2008.

[7] Dharmapurikar S, Paxson V. Robust TCP stream reassembly in the presence of adversaries. In: Proc. of the 14th Conference on USENIX Security Symposium. Baltimore, Maryland, USA, 2005.

[8] Schuehler D V, Lockwood J W. A modular system for FPGA-based TCP flow processing in high-speed networks. In: Proc. of the 14th International Conference on Field Programmable Logic and Application. Leuven, Belgium, 2004.

[9] Schuehler D V, Lockwood J W. TCP-splitter: A TCP/IP flow monitor in reconfigurable hardware. In: Proc. of the 10th Symposium on High Performance Interconnects. Stanford University, California, USA, 2002.

[10] Dharmapurikar S, Krishnamurthy P, Sproull T, LockWood J. Deep packet inspection using parallel bloom filters. In: Proc. of the 11th Symposium on High Performance Interconnects. Stanford University, California, USA, 2003.

[11] Moscola J, Lockwood J, Loui R P, Pachos M. Implementation of a content-scanning module for an internet firewall. In: Proc. of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). Napa, California, 2003.

[12] Shen J, Zhou X S, Zhang F, Yu Z Y. Optimized design and research of firewall based on network processor. *Computer Engineering*, 2007, **33**(10): 172-174. (in Chinese).

[13] Zhong T, Liu Y, Li Z J, Qin Z G. Research of a comprehensive IPv4/IPv6 firewall system based on network processor. *Journal of Communications*, 2006, **27**(2): 14-20. (in Chinese).

[14] Xu B, Zhou X, Li J. Recursive shift indexing: A fast multi-pattern string matching algorithm. In: Proc. of the 4th International Conference on Applied Cryptography and Network Security (ACNS). Singapore: Springer-Verlag, 2006.

[15] http://www.snort.org/, 2008.