Towards High-performance Flow-level Packet Processing on Multi-core Network Processors

Yaxuan Qi[1, 3], Bo Xu[1, 2], Fei He[1, 2], Baohua Yang[1, 2], Jianming Yu[1, 2] and Jun Li[1, 3]

[1]Research Institute of Information Technology, Tsinghua University
[2]Department of Automation, Tsinghua University
[3]Tsinghua National Lab for Information Science and Technology
{yaxuan, junl}@tsinghua.edu.cn

ABSTRACT

There is a growing interest in designing high-performance network devices to perform packet processing at flow level. Applications such as stateful access control, deep inspection and flow-based load balancing all require efficient flow-level packet processing. In this paper, we present a design of high-performance flow-level packet processing system based on multi-core network processors. Main contribution of this paper includes: a) A high performance flow classification algorithm optimized for network processors; b) An efficient flow state management scheme leveraging memory hierarchy to support large number of concurrent flows; c) Two hardware-optimized order-preserving strategies that preserve internal and external per-flow packet order. Experimental results show that: a) The proposed flow classification algorithm, AggreCuts, outperforms the well-known HiCuts algorithm in terms of classification rate and memory usage; b) The presented SigHash scheme can manage over 10M concurrent flow states on the Intel IXP2850 NP with extremely low collision rate; c) The performance of internal packet order-preserving scheme using SRAM queue-array is about 70% of that of external packet order-preserving scheme realized by ordered-thread execution.

Categories and Subject Descriptors

C.2.0 [**Computer Communication Networks**]: General Security and protection (e.g., firewalls); C.4 [**Performance of Systems**]: Design studies.

General Terms

Algorithms, Experimentation, Performance

Keywords

Classification, Hashing, Packet Order, Network Processor

ANCS'07, December 3–4, 2007, Orlando, Florida, USA.
Copyright 2007 ACM 978-1-59593-945-6/07/0012...$5.00.

1. INTRODUCTION

The continual growth of network communication bandwidth and the increasing sophistication of types of network traffic processing have driven the need for designing high-performance network devices to perform packet processing at flow level. Applications such as stateful access control in firewalls, deep inspection in IDS/IPS, and flow-based scheduling in load balancers all require flow-level packet processing. Basic operations inherent to such applications include:

♦ **Flow classification:** Flow-level packet processing devices are required to classify packets into flows according to a classifier and process them differently. As the new demand for supporting multiple services (voice, video, and data) arises, the workload for such devices to perform fast flow classification becomes much heavier. Therefore, it is challenging to perform flow classification at line speed on these network devices.

♦ **Flow state management:** Per-flow states are maintained in order to correctly perform packet processing at a semantic level higher than the network layer. Such stateful analysis brings with it the core problem of state management: what hardware resources to allocate for holding states and how to efficiently access them. This is particularly the case for in-line devices, where flow state management can significantly affect the overall performance.

♦ **Per-flow packet order-preserving:** Another important requirement for networking devices is to preserve packet order. Typically, order-preserving is only required between packets on the same flow. Although there have been some order-preserving techniques for traditional switch architecture, flow-level packet order-preserving for parallel switches still remains as an open issue and motivates the research today.

Traditionally flow-level packet processing devices rely on ASIC/FPGA to perform IP forwarding at line-rate speed (10Gbps) [12] [23] [24]. As the network processor (NP) emerges as a promising candidate for a networking building block, NP is expected to retain the same high performance as that of ASIC and to gain the time-to-market

Table 1: Hardware overview of IXP2850

Intel XScale core:	Each IXP2850 includes an XScale core. The Intel XScale core is a general purpose 32-bit RISC processor.
Multithreaded microengines	The IXP2850 network processor has 16 MEs working in parallel on the fast path, running at 1.4 GHz clock frequency.
Memory hierarchy	IXP2850 has up to 4 channels of QDR SRAM running at 233 MHz and 3 channels of RDRAM running at 127.3 MHz.
Build-in media interfaces	IXP2850 has flexible 32-bit media switch interfaces. Each interface is configurable as media standard SPI-4 or CSIX-L1 interfaces.

advantage from the programmable architecture [16]. In this paper, we design a 10-Gbps flow-level packet processing system based on a typical multi-core network processor. Main contributions of this paper are:

♦ **An NP-optimized flow classification algorithm:** The algorithm we proposed in this paper, Aggregated Cuttings (AggreCuts), is based on the well-known HiCuts algorithm, while optimized for multi-core network processors. Different from HiCuts, AggreCuts has explicit worst-case search time. To avoid burst of the memory usage, AggreCuts also adopts a hierarchical space aggregation technique that significantly reduces the memory usage.

♦ **An efficient flow state management scheme:** The proposed scheme, Signature-based Hashing (SigHash), can support more than 10M concurrent flow states with low collision rate. Different from existed hashing table design, signatures for collision-resolving are stored in word-oriented SRAM chips in SigHash, while their corresponding flow states are stored in burst-oriented DRAM chips. Consequently, the flow state update speed at near line rate is guaranteed, and the large-storage requirement for millions of flows is met as well.

♦ **Two hardware-supported packet order-preserving schemes:** The first scheme is external packet order-preserving (EPO), which guarantees that packets leave the device in the same order in which they arrived. The second scheme is internal packet order-preserving (IPO), which not only enforces packet order in each flow, but also guarantees that packets belong to the same flow are processed by the same thread. When implemented on NP, EPO uses ordered-thread execution technique to maintain external packet order, while IPO employs the SRAM queue array to achieve internal packet order-preserving.

Experimental results on the Intel IXP2850 NP show that:

♦ **AggreCuts outperforms the existing best-known algorithm HiCuts in terms of both classification rate and memory usage.** AggreCuts uses an order of magnitude less memory than HiCuts, while guarantees explicit worst-case search time. When tested with minimum Ethernet packets against real-life rule sets, the worst-case throughput of AggreCuts reaches 9-Gbps on the IXP2850 NP.

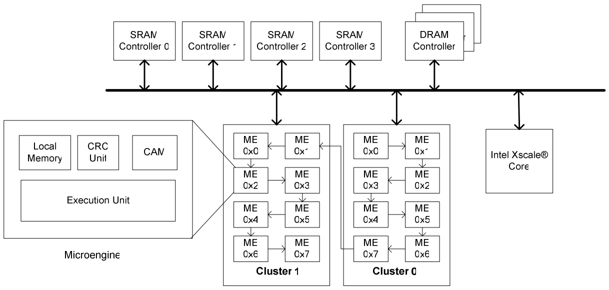

Figure 1. Architecture of the IXP2850 NP

♦ **The presented SigHash scheme can support more than 10M concurrent flow states on IXP2850 NP, which is over 20 times more than traditional hash implementation.** When tested with minimum Ethernet packets flows, SigHash achieves 10-Gbps state update rate.

♦ **The performance of IPO scheme using SRAM queue-array is close to that of EPO realized by ordered-thread execution.** Experimental results with real-life traffic also show that even the simple direct hashing scheme used by thread allocation is well-suited for our IPO implementation.

This paper is organized as follows. In Sections 2, backgrounds of network processor and related work are described. AggreCuts for flow classification and SigHash for flow state management are presented in Section 3 and Section 4. In Section 5, two packet order-preserving schemes, EPO and IPO are described. Experimental results are discussed in Section 6. As a summary, in Section 7, we state our conclusion.

2. BACKGROUND AND RELATED WORK

In this section, we first introduce the Intel IXP2850 network processor, and then discuss related work.

2.1 The Intel IXP2850 Network Processor

2.1.1 Architecture of IXP2850

Network processors are typically characterized as distributed, multi-processor, multi-threaded architectures designed for hiding memory latencies in order to scale up to very high data processing rates. The architecture of Intel IXP2850 is motivated by the need to provide a building block for 10-Gbps packet processing applications. A simplified block diagram and its description of the Intel IXP2850 NP are shown in Figure 1 and Table 1 respectively [17-20].

2.1.2 Programming Challenges

Network processing applications are targeted at specific data rates. In order to meet these throughput requirements, NPs must complete the packet processing tasks on a given packet before another packet arrives. To keep up with the back-to-back arrival of minimum size packets at line rate, an NP faces the following programming challenges [21]:

- **Achieving a deterministic bound on packet processing operation:** Due to the line rate constraint, we need to design the network algorithms in such a way that the number of clock cycles to process the packet on each microengine (ME) does not exceed an upper bound.
- **Masking memory latency through multi-threading:** Even if the packet can be processed within a definite time interval, it is not sufficient to meet the line rate processing requirements because memory latencies are typically much higher than the amount of processing budget.
- **Preserving packet order in spite of parallel processing:** Another significant challenge in programming the NPs is to maintain packet order. This is extremely critical for applications like media gateways and traffic management.

In the following sections, we will address these issues by employing hierarchical memories, parallel processing and efficient data-structures to achieve optimized packet processing performance on the Intel IXP2850 NP.

2.2 Related Work on NP

D. Srinivasan and W. Feng implemented the basic Bit-Vector flow classification algorithm on the Intel IXP1200 network processor [23]. They compared parallel mapping and pipelined mapping of the algorithm and showed that parallel mapping has better processing rate. The limitation of their work is that the implemented Bit-Vector algorithm can only support 512 rules on IXP1200, which is too small compared to current real-life rule sets [10]. Recent work presented by D. Liu et al. proposed a modified recursive flow classification (RFC [7]) algorithm, named Bitmap-RFC, which reduces the memory usage of RFC by applying a bitmap compression technique [13]. Experimental results showed that the Bitmap-RFC algorithm achieves 10-Gbps speed on the Intel IXP2800 NP. Although Bitmap-RFC obtained high throughput on NP, it still requires large memory storage compared to other algorithms such as HiCuts and HyperCuts. Moreover, the total amount of memory words loaded from SRAM is 5 times larger than that of the original RFC algorithm. This will potentially affects the performance of other applications implemented on the same NP.

Flow state management is becoming more critical for network devices to guarantee semantic-level security. Leading security enterprises like CheckPoint have been developing there own stateful firewalls, which provide deep inspection into the network traffic flows rather than scanning the packets individually. Besides, flow state management is also a compulsory segment for TCP reassembly [26] [27]. Although the basic techniques used for flow state management, such as exact match by hashing, have been well-studied [28], however, how to efficiently implement flow state management on multi-core network processor still remains as an open issue. To address this issue, two problems must be well resolved: what hardware resources to allocate for holding states and how to efficiently update them. This is particularly the case for in-line devices, where flow state management can significantly affect the overall performance.

Another important requirement for networking devices is to preserve packet order. Typically, packet order-preserving is only required between packets on the same flow that are processed by parallel processing engines. In network devices processing at network layer, the external packet order-preserving (EPO) scheme is sufficient, but applications that process packets at semantic levels require internal packet order-preserving (IPO), in which packets belong to the same flow are handled by the same thread [26]. Generally, the EPO scheme exploits greater-degree of concurrency and is expected to achieve finer-grain distribution of workload across microengines than the IPO scheme [33]. However, the EPO scheme also might incur higher overhead for accessing per-flow state, since with this scheme the concurrent access to the shared flow states by multiple threads must be synchronized using inter-thread signals and memory-locks.

3. FLOW CLASSIFICATION

There are a number of network services that require flow classification, such as policy based routing, stateful access-control, differentiated qualities of service, and traffic billing. In each case, it is necessary to determine which flow an arriving packet belongs to in order to determine whether to forward it, what class of service it should receive, or how much should be charged for transporting it. In this section, we present a high-performance flow classification algorithm optimized for the multi-core network processor.

3.1 Algorithm Selection

Flow classification has been proved to be theoretically hard, and hence it is impossible to design a single algorithm that performs well for all cases [5]. Fortunately, real-life flow classification rule sets have inherent characteristics that can be exploited to reduce the temporal and spatial complexity. In existing literatures, a variety of characteristics of real-life rule sets were presented and exploited in designing efficient flow classification algorithms [3] [7] [8] [9] [10]. All these algorithms can be categorized as field-independent search and field-dependent search [11]:

- **Field-independent search**: Algorithms such as RFC [7] and HSM [8] perform independent parallel searches on indexed tables; the results of the table searches are combined in multiple phases to yield the final classification result. All the entries of a lookup table are stored consecutively in memory. The indices of a table are obtained by space mapping and each entry corresponds to a particular sub-space and stores the search result at current stage. Although these sorts of algorithms are very fast in term of classification speed, they might require large memories to store the big cross-producting tables.

♦ **Field-dependent search**: HiCuts [3] and HyperCuts [9] are examples of algorithms employing field-dependent searches, i.e., the results of fields that have already been searched determine the way in which subsequent fields will be searched. The main advantage of this approach is that the intelligent and relatively simple decision-tree classifier can be used. Although in most cases, decision-tree algorithms require less memory than field-independent search algorithms [12], they tend to result in implicit worst-case search time and thus cannot ensure a stable worst-case classification speed.

Because the large memory usage of field-independent search algorithms can hardly be satisfied by current SRAM chips on network processors [13], in this paper, the proposed algorithm is based on HiCuts, one of the best-known field-dependent search algorithms.

3.2 Reduce Memory Accesses

Although HiCuts has good time/space tradeoffs and works well for real-life rule sets, the straightforward implementation of HiCuts without any NP-aware optimization on an multi-core NP suffers from:

♦ **Non-deterministic worst-case search time:** Because the number of cuttings varies at different tree nodes, the HiCuts tree may have non-deterministic worst-case depth. Thus, although worst-case search time is the most important performance metric in packet classification applications, HiCuts does not have an explicit bound for search.

♦ **Excessive memory access due to linear search:** Although the number of rules for linear search is limited to 4~16 in HiCuts, it still requires tens of memory accesses to off-chip SRAM chips. Experimental results show that linear search is very time-consuming on NPs.

Thus an ideal decision-tree based flow classification algorithm optimized for network processors must have explicit worst-case bounds for search and avoid the linear search at leaf-nodes. Our motivations to design such an algorithm are:

♦ **Fix the number of cuttings at internal-nodes:** If the number of cuttings is fixed to 2^w (w is a constant referred as *stride*), the current search space is then always segmented into 2^w sub-spaces at each internal-node. This guarantees a worst-case bound of $O(W/w)$, where W is the bit-width of the packet header.

♦ **Eliminate linear search at leaf-nodes:** Linear search can be eliminated if we "keep cutting" until every sub-space is full-covered by a certain set of rules. The rule with the highest priority in the set is then the final match.

Consider the common 5-tuple flow classification problem, where W=104. If w is fixed to 8, then the worst-case search time (memory access) is limited to 104/8=13, and in this case, no linear search is required because in those longest tree-paths, every bit has already been "cut".

3.3 Compress the Data Structure

Admittedly, both the fixed stride and the elimination of linear search tend to result in memory burst. Therefore, the main optimization task becomes how to effectively reduce the memory storages. In the previous storage optimization scheme, HiCuts maximizes the reuse of child nodes by employing pointer arrays to aggregate sub-spaces with identical rule sets. However, the use of pointer arrays will also require large amount of memory because the size of each array is considerably large when the number of cuttings is fixed. For example, if w is fixed to 8, each internal-node will store 256 pointers to link to child-nodes. Typically, a decision-tree contains tens of thousands internal-nodes, then the total memory required to store the pointer arrays will exceed tens of mega-bytes, which is too large for current SRAM chips [18].

To effectively reduce the memory usage of these pointer arrays, some exiting work [11] [13] [22] [29] employ the bit-compression technique to aggregate consecutive pointers: First, use an Aggregation Bit String (ABS) to track the appearance of unique elements in the pointer array, and then compress a sequence of consecutively identical pointers as one element in a Compressed Pointer Array (CPA). More specifically, each bit of an ABS corresponds to an entry in the original pointer array, with the least significant bit corresponding to the first entry. A bit in an ABS is set to '1' if its corresponding entry in the original pointer array is different from its previous one, i.e. bit set in an ABS indicates that a different sequence of consecutively identical pointer starts at the corresponding position. Whenever a bit is set, its corresponding unique pointer is appended in the CPA. Accordingly, the n-th pointer in the original point array can be found by: first adding the first n bits in the ABS to get an index, and then use the index to load the pointer from CPA.

Ideally, all the pointer arrays should be compressed using ABS and CPA. However, loading such a bit-string may also cause excessive memory accesses. Assume w=8, the size of the ABS is thus (256 bits)/32=8 32-bit long-words. Therefore at each internal-node, we have to load 8 long-words from the off-chip SRAM. Since the tree-depth is 13, the overall memory accesses to classify a single packet is more than 13*8=104 long-words, which is too much for a practical memory bandwidth budget to reach multi-Gbps packet classification rate [1] [20].

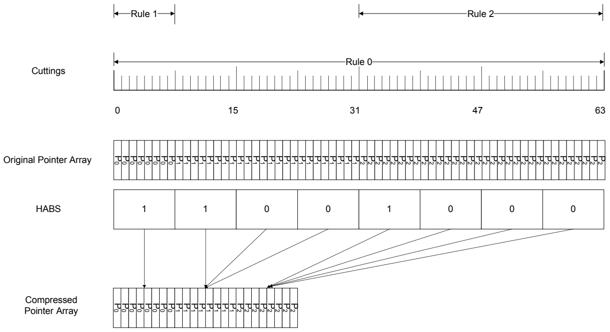

Figure 2. Hierarchical Aggregation Bit-String

Fortunately, in our experiments on a variety of real-life rule sets, we found that the number of child nodes of a certain internal tree node is commonly very small: with 256 cuttings at each internal-node, the average number of child nodes is less than 10. This observation is very consistent to the results reported in other related work [3] [9] [10]. Such small number of child nodes indicates that the pointer array is very sparse, i.e. the number of bits set in ABS is also sparse. Thus, this observation motivates us to further compress the ABS to effectively reduce the number of memory accesses.

Figure 2 illustrates how to use the Hierarchical Aggregation Bit-String (HABS) [29] to further compress the data structure. Define the size of HABS as 2^v, the number of pointers as 2^w, and u=w-v. To compress the 2^w pointers: First, divide the 2^w pointers into 2^v sub-arrays. Then set the bits in HABS to '1' if the 2^u consecutive pointers in its corresponding sub-array are different from the pointers in previous sub-array, i.e. a bit set in an HABS indicates that a different sequence of consecutively identical sub-array of pointers starts at the corresponding position. On the same time, whenever a bit is set, its corresponding sub-array of pointers is appended in the CPA. According to this scheme, the n-th pointer in the original point array can be located by: 1) extract the higher v bits of n to get a v-bit value m; 2) extract the lower u bits of n to form a u-bit value j; 3) add 0~m bits of the HABS to get a sub-array index i; 4) use $((i<<u)+j)$ as the index to load the corresponding pointer from CPA. Different from previous work [29], in the implementation of AggreCuts, the size of HABS is set to 8 and HABS is stored together with the cutting information and the next-node address base within a single 32-bit long-word (see Table 2). Such a data-structure can be effectively accessed by the word-oriented SRAM controller on IXP2850 and the computation of HABS can be done within 3 cycles using the POP_COUNT instruction [24]. The overall data-structure of a sample decision-tree built by AggreCuts is depicted in Figure 3.

4. FLOW STATE MANAGEMENT

Flow state management experiences a large number of updates over a short period of time as new sessions are initiated, and old sessions are closed down. Current session tables to store flow states can also be exceptionally large, on the order of 1 million entries. Hashing algorithms are well-suited for exact match problems and thus widely used for flow state management. A hash table is made up of two parts: an array and a mapping function. The array is used as a table to store the data and the mapping function, also known as the hashing function, is used to convert the input space into array indices. In this section, we propose an efficient hashing scheme to implement flow state management on multi-core network processors.

Table 2. Compact Tree Node Structure

Bits	Description	Value

31:30	dimension to Cut (d2c)	d2c=00: src IP; d2c=01: dst IP;
		d2c=10: src port; d2c=11: dst port.

29:28	bit position to Cut (b2c)	b2c=00: 31~24; b2c=01: 23~16
		b2c=10: 15~8; b2c=11: 7~0

27:20	8-bit HABS	if w=8, each bit represent 32 cuttings; if w=4, each bit represent 2 cuttings.
19:0	20-bit Next-Node CPA Base address	The minimum memory block is 2^w/8*4 Byte. So if w=8, 20-bit base address support 128MB memory address space; if w=4, it supports 8MB memory address space.

Note: We use d2c=11, b2c=00 to indicate the cutting of the 5th dimension (the 8-bit transport layer protocol).

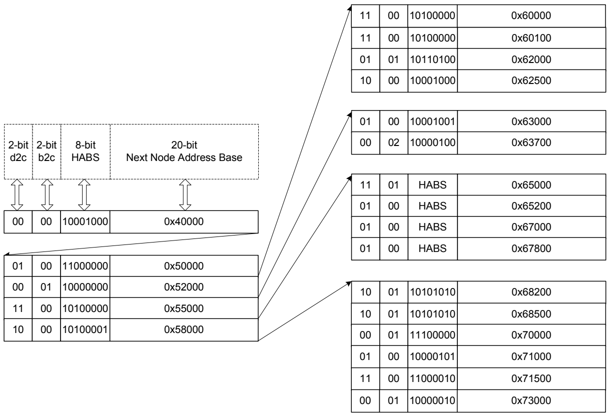

Figure 3. Data-structure of a Sample AggreCuts Tree.

4.1 Hash Function Selection

Hash functions are used to convert the input of width W bits into a hash key of width N bits. To reduce the probability of collisions, it is important to select an appropriate hash function. In this paper, we compare three typical hash functions: FNV Hash [15], Jenkins Hash [16] and CRC Hash [31]. The evaluation packet set is real flow packets collected at edge firewalls of large enterprises. The

Table 3. Evaluation of Typical Hash Functions

Hash	Collision Rate			
Function	l=1:1	l=1:2	l=1:4	l=1:8
CRC	0.368670	0.107371	0.028602	0.007501
FNV	0.367435	0.106284	0.028365	0.007384
Jenkins	0.368289	0.106969	0.028541	0.007563

five-tuple fields, i.e. source IP, destination IP, source port, destination port and transport layer protocol, are concatenated to form the 104-bit hash input. The load factor *l* is defined as *n/m*, where *m* is the total number of buckets in the hash table and *n* is the maximum number of buckets that are occupied at the same time.

Table 3 shows the collision rate of the three typical hash functions. The test is done with different load factor *l*, ranging from 1:8 to 1:1. The collision rate is defined as the number of collisions divided by the total number of input packets. From Table 3, it is clear that the performance of the three different hashing algorithms is very close to each other. Because each microengine on IXP2850 has an on-chip CRC unit, which provides very fast CRC computation [24], we choose to use the CRC as the hashing function.

4.2 Hash Table Design

The data-structure of hash table is of great importance to the design of efficient flow state management, which takes the advantage of the parallelism of multi-core and multi-threaded network processor. The hash table data structure explains how we organize the flow state entries.

The traditional hash scheme, which is named DirectHash in this paper, stores flow states in a single hash table indexed by the hashing value and uses link lists to handle hash collisions. Each entry in the link list stores a full flow state consisting of the 5-tuple header, sequence/acknowledge number, and other flow information according to different applications. When collision occurs, the link list entries will be checked one by one to find out the exact match. Although DirectHash is simple to implement on NP, this scheme suffers from:

♦ **SRAM Size Limit:** Assuming that 10M concurrent flow states were to support with a load factor of 1:1, and each flow state entry is 32 bytes in size, the total size of the hash table by DirectHash will be 320MB, which is too large compared to the 24MB SRAM chips on IXP2850.

♦ **Excessive Memory Accesses:** According to our experiments (see Table 3), with load factor of 1:1, the hash collision rate will be greater than 30%, which means more than 30% of packets will be stored in the link list, and thus results in excessive memory accesses.

To address these problems, we employ the idea of hierarchical hashing [1] and propose a signature-based hashing scheme named SigHash. As shown in Figure 4, the SigHash uses two hash tables: the signature table in SRAM and the flow state entry table in DRAM. Both the tables are indexed using 24 lower order bits of the hash value returned by the hash function. Therefore, both the primary and secondary tables contain 2^{24} buckets. Each *bucket* is further organized as 4 bins. Each *bin* corresponds to a rule that has been inserted into the hash table. The primary table is a compact table that stores hash *signatures* in each bin. A

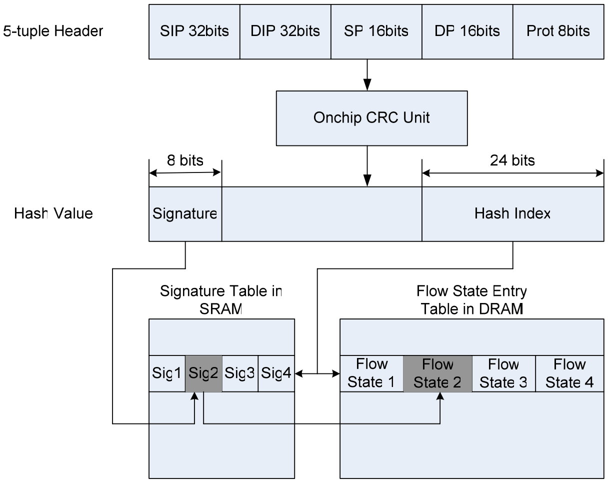

Figure 4. Data-structure of SigHash

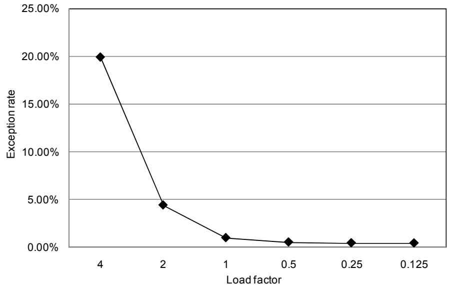

Figure 5. Load Factor Selection for SigHash

signature is a contiguous set of 8 bits taken from the hash index returned by the hash function. These 8 bits are distinct from the set of 24 bits used as an index into the two tables. In comparison to DirectHash, SigHash is more efficient because:

♦ The relatively high speed memory usage of SRAM is significantly reduced by the hierarchical memory management. *n* concurrent flows with load factor *l* only need *n/l**4 Bytes SRAM memory storage, together with *n/l*sizeof(flow state)* Bytes DRAM.

♦ Assisted by the signature checking in SRAM, only one SRAM access is required if the number of collisions are less than 5. To check out whether the incoming flow matches the corresponding flow state in DRAM, additional one DRAM access is required. Thus, SigHash always provides deterministic performance when there are four or less collisions

♦ If the collision entries exceed 5, the packet will be sent to the slow path and be treated as an exception in XScale core. However, experimental results with real-life traffics [5] indicate that, with an appropriate choice of load factor, the number of exceptional packets can be very small. From Figure 5 we can see that, when the load factor is smaller than 1, the exception rate is less than 2%.

5. PER-FLOW PACKET ORDER-PRESERVING

Another important requirement for networking devices is to maintain packet order. Typically, ordering-preserving is only required between packets on the same flow. In network devices processing at network layer, the external packet order-preserving (EPO) scheme is sufficient, but applications that process packets at semantic levels require internal packet order-preserving (IPO), in which packets belong to the same flow are processed by the same thread. In this section, we discuss how to efficiently implement EPO and IPO in the Intel IXP 2850 network processor.

5.1 External Packet Order-preserving

5.1.1 Ordered-thread Execution

EPO can be realized by ordered-thread execution strategy [1]. In this design, every dispatch loop assumes that the packets are in order when it receives them from the previous processing state. It uses an ordered critical section to read the packet handles off the scratch ring from the previous state. The threads then process the packets, which may get out of order during packet processing. At the end of the dispatch loop, another ordered critical section ensures that the threads write the packet handles to the scratch ring in the correct order. The implementation of ordered critical sections typically uses strict thread ordering enforced via inter-thread signaling.

5.1.2 Mutual Exclusion by Atomic Operation

In the EPO scheme, packets belong to the same flow may be allocated to different threads to process, thus mutual exclusion is needed because multiple threads may access the same flow state entry in the hash table. Mutual exclusion can be implemented by locking. The Intel IXP2850 NP provides an efficient way to implement a lock using SRAM atomic instructions such as *sram_test_and_set()*. In our design, once a thread is accessing the session entry data structure, the MUTEX lock bit is set by an atomic operation so that any other thread cannot access the same session entry. At the end of the packet processing, the thread clear the MUTEX lock bit so that other threads can access the shared memory.

5.2 Internal Packet Order-preserving

5.2.1 Internal Packet Order-preserving via SRAM Q-Array

IPO requires that packets belong to the same flow should be processed by the same thread. This requirement can be met using multiple memory queues, each of which corresponds to a certain thread. The IXP2850 hardware supports a 64-element SRAM Queue Array per SRAM channel, providing fast access to these queues through a 64-element hardware cache in the SRAM controller. Figure 6 shows how to implement IPO using SRAM Queue Array: Packets belong to the same flow (with the same CRC hash value) will first be put into the same SRAM Queue, and then be processed by the corresponding thread.

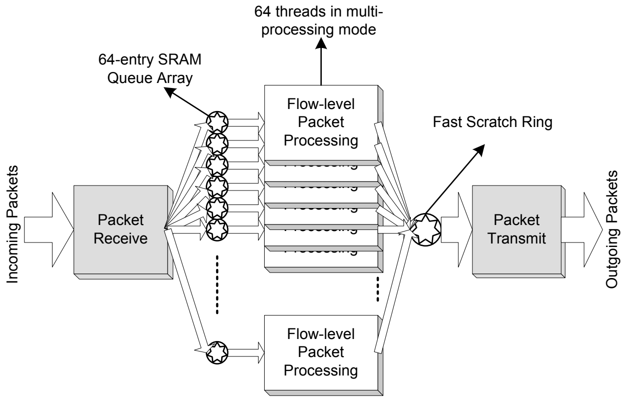
64 threads in multi-processing mode
64-entry SRAM Queue Array
Flow-level Packet Processing
Fast Scratch Ring
Incoming Packets
Packet Receive
Packet Transmit
Outgoing Packets
Flow-level Packet Processing

Figure 6. Internal Packet Order-preserving via Sram Queue Array

5.2.2 Workload Allocation by CRC Hashing

In this paper, a pure hash-based workload allocation module is implemented for IPO. With direct hashing, the load balancing module applies a hash function to a set of fields of the five-tuple, and uses the hash value to select the outgoing queue in the SRAM Queue Array. In our study, two hashing schemes are evaluated: hashing of the destination address ($H = DestIP \bmod N$) and hashing of the five-tuple packet header using CRC (H = CRC32 (five-tuple) mod N). We found that CRC hashing scheme performs much better than the destination address hashing over different real-life traffic traces. Such results are consistent to the evaluations of Cao's work [31]. Therefore, we choose the CRC hashing as our workload allocation scheme.

Note that W. Shi, M. H. MacGregor and P. Gburzynski have proved that pure hashing is not able to well balance the workload due to the Zipf-like distributions of flow-size in real-life traffic [32]. However, in practice, the performance of hash-based load balancing module is good enough (see the experiments in Section 6.4.1)

6. PERFORMANCE EVALUATION

6.1 Development Kits and Test Environments

There are two basic programming choices in the Intel Software Developer Kit (Intel SDK): programming in assembly language (Microcode) or programming in C language (MicroC). To make better compatibility with Intel SDK, and to avoid dependency on compiler optimizations, all the application and algorithms are developed using Microcode assembly with the software framework in Intel SDK4.0 [25]. To evaluate the performance, the application is tested and run in the IXP2850 Developer Workbench, which offers a cycle-accurate simulator of the IXP2850 NPs. This environment provides access to several performance metrics that reflect the actual IXP2850

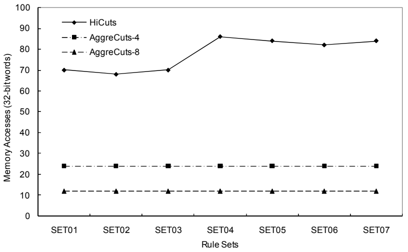

Figure 7. Worst-Case Memory Access

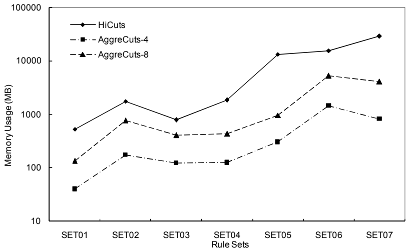

Figure 8. SRAM Usage Evaluation

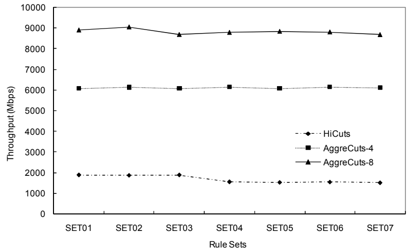

Figure 9. NP Throughput

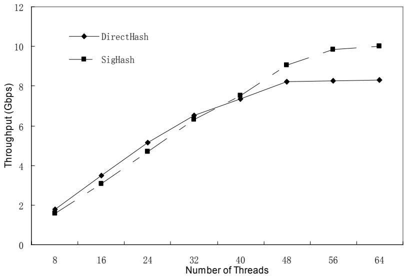

Figure 10. DirectHash vs. SigHash

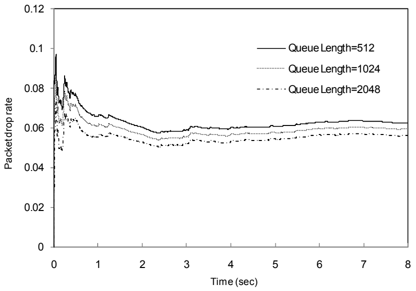

Figure 11. Workload Allocation

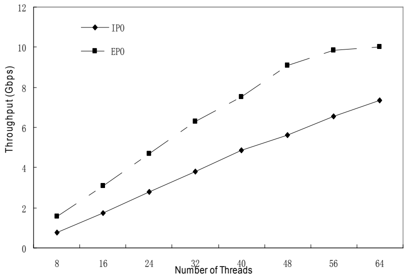

Figure 12. EPO vs. IPO

hardware. The application is also tested on a dual-IXP2850 platform to ensure the code accuracy and compatibility on hardware.

6.2 Performance of AggreCuts

We compare three main performance metrics of the AggreCuts to the HiCuts algorithm: the worst-case memory access, total memory usage and throughput on IXP2850. The testing rule sets, denoted as SET01~SET07, are all real-life 5-tuple ACLs obtained from large enterprises [30]. The size of these rule sets ranges from 68 (SET01) to 1530 (SET07). AggreCuts-4 refers to *w*=4, i.e. at each internal node, the current search space is cut into 2^4 sub-spaces. Similarly, AggreCuts-8 refers to *w*=8.

6.2.1 Worst-case Memory Access

Worst-case memory access, indicating the worst-case classification speed, is the most important performance metric in evaluation of a flow classification algorithm. From Figure 7 we can see that the worst-case memory accesses of AggreCuts-8 and AggreCuts-4 are less than 1/6 and 1/3 of that of HiCuts respectively. This is because the worst-case tree depth of HiCuts depends on the data-structure of the rule set, while that of AggreCuts is set-independent due to the fixed stride cutting scheme. Such a definite worst-case memory access is expected to guarantee stable performance of high-speed flow classification on the network processor.

6.2.2 Memory Usage

The memory usages of AggreCuts-4, AggreCuts-8 and HiCuts against the 7 real-life rule sets are shown in Figure 8. It can be seen from the figure that: AggreCuts-4 uses nearly an order of magnitude less memory than HiCuts. AggreCuts-8 also has much better spatial performance than HiCuts. More specifically, AggreCuts-4 uses less than 1.4MB memory against all the 7 rule sets, and AggreCuts-8 uses less than 5.3MB memory, both of which are less than the size of a single chip of SRAM on the IXP2850 network processor (there are three 8MB SRAM chips on IXP2850). In comparison, the memory usage of HiCuts is larger than 28MB, which is larger than the total amount of all the three chips of SRAM.

6.2.3 Throughput on NP

To evaluate the throughput of the worst-case performance on IXP2850, we use minimum 64Byte Ethernet packets as the input traffic and set each packet to match the longest tree depth (i.e. each packet will incur the worst-case memory access). Figure 9 shows the throughput achieved by AggreCuts-4, AggreCuts-8 and HiCuts. From this figure, we see that AggreCuts-8 reaches nearly 9-Gbps throughput and AggreCuts-4 also has a stable 6-Gbps performance. In comparison, the throughput of HiCuts is less than 2Gbps, and as the number of rules increases, its performance decreases.

6.3 Performance of SigHash

Two hash schemes are implemented on the platform of the Intel IXP2850 NP: the DirectHash scheme based on SRAM and the SigHash scheme based on SRAM as well as DRAM. From Figure 10, we can see that the DirectHash scheme reaches 8.3-Gbps throughput with 64 threads, while the SigHash scheme reaches 10-Gbps line speed. The figure also shows that when the number of threads exceeds 40, the performance of DirectHash does not increase linearly. This is because the SRAM READ COMMAND FIFO becomes a bottleneck as more and more threads issues memory access commands concurrently. In contrast, the SigHash scheme takes the advantage of the DRAM interleaving storage mechanism which evenly distributes consecutive memory access into three channels to fast support burst-oriented memory access. Moreover, with 24MB SRAM and 2GB DRAM on the IXP2850 NP, the SigHash scheme can support over 10M concurrent sessions while the DirectHash scheme can only maintain less than 200K concurrent flows.

6.4 Performance of Packet Order-preserving

6.4.1 Workload Allocation by CRC Hashing

To evaluate the impact of IPO on the utilization of microengines caused by the load distribution fluctuation, simulation is performed on the CRC hashing module, using packet traces (CESCA-I) from NLANR PMA [34].

In this simulation, packets are distributed into 64 queues, and the length of each queue is set to be 512, 1024 and 2048. The total processing capacity of the 64 threads is assumed to match the input packet rate. As shown in Figure 11, tested with three different queue lengths, the average packet drop rates caused by the queue overflow are below 6%, i.e. the utilization of microengines is higher than 94%. This also indicates that, on average, there are 64*6%=4 threads working in idle time.

6.4.2 EPO vs. IPO

The EPO scheme is implemented by ordered-thread execution and the IPO scheme is realized through SRAM queue array. Figure 12 shows that, the EPO scheme reaches the line speed of 10-Gbps with 64 threads, while the IPO scheme achieves its maximum throughput of 7.4-Gbps. The reason why IPO scheme runs at a lower rate mainly contains two aspects: First, the IPO scheme forces each flow to be processed in a specified thread. This flow-level workload distribution brings in the non-uniformity in load balancing. Secondly, the flow-level workload allocation incurs additional processing overhead, especially the memory access to load packet headers for CRC computation. Nevertheless, the IPO scheme is expected to achieve higher performance if more threads could be allocated, since it can be seen from Figure 12 that the throughput of the IPO scheme still grows linearly when running on 64 threads.

7. CONCLUSION

In this paper, we present a high-performance flow-level packet processing system based on multi-core network processors. At first, a high performance flow classification algorithm optimized for network processors is proposed that outperforms the existing best-known HiCuts algorithm. Secondly, an efficient flow state management scheme using signature-based hashing is presented, which can support more than 10M concurrent flows on the IXP2850 network processor and reach 10-Gbps line speed. In addition, two hardware-supported packet order-preserving strategies that preserve internal and external packet orders respectively are implemented and evaluated on the IXP2850 NP. Experimental results show that the performance of internal packet order-preserving scheme using SRAM queue-array is close to that of external packet order-preserving scheme realized by ordered-thread execution.

In future work, we plan to implement TCP stream reassembly and pattern matching building blocks on the IXP2850 NP. Although both tasks are hard due to deeper packet content to inspect and more complicated states to maintain, they can be implemented as degenerated or simplified versions [26] [27]. Our future work also includes the implementation of flow-level load balancing [31-33] and application-level flow detection [35-37] on the IXP2850 NP. Note that all these work in our future work are based on the proposed flow-level packet processing systems. We believe that, as the continual growth of network traffic rate and the increasing sophistication of types of network traffic processing, more and more complicated network applications will emerge using parallel processing network devices to perform high-speed packet processing at flow level.

8. ACKNOWLEDGEMENTS

The authors would like to thank anonymous reviewers for their valuable comments. This work is supported in part by National 863 Hi-tech Project of China.

9. REFERENCES

[1] U. R. Naik and P. R. Chandra, “Designing High-performance Networking Applications”, Intel Press, 2004.

[2] T. Sherwood, G. Varghese, and B. Calder, “A Pipelined Memory Architecture for High Throughput Network Processors,” Proc. of the 30th International Symposium on Computer Architecture, 2003.

[3] P. Gupta and N. McKeown, “Packet Classification Using Hierarchical Intelligent Cuttings”, Proc. Hot Interconnects, 1999.

[4] P. Gupta and N. McKewon, “Algorithms for Packet Classification”, IEEE Network, March/April, 2001.

[5] M. H. Overmars and A. F. van der Stappen, “Range Searching and Point Location among Fat Objects”, Journal of Algorithms, 21(3), 1996.

[6] Y. Qi, B. Xu and J. Li, “Evaluation and Improvement of Packet Classification Algorithms”, Proc. of the 1st International Conference on Network and Services (ICNS), 2005.

[7] P. Gupta and N. McKeown, “Packet Classification on Multiple Fields”, Proc. ACM SIGCOMM, 1999.

[8] B. Xu, D. Jiang and J. Li, “HSM: A Fast Packet Classification Algorithm”, Proc. of the 19th International Conference on Advanced Information Networking and Applications (AINA), 2005.

[9] S. Singh, F. Baboescu, G. Varghese and J. Wang, “Packet Classification Using Multidimensional Cutting”, Proc. of ACM SIGCOMM, 2003.

[10] M. E. Kounavis, A. Kumar, H. Vin, R. Yavatkar and A. T. Campbell, “Directions in Packet Classification for Network Processors”, Proc. of the 2nd Workshop on Network Processors (NP2), 2003.

[11] J. van Lunteren and T. Engbersen, “Dynamic Multi-Field Packet Classification”, Proc. of IEEE GLOBECOM, 2002.

[12] D. E. Taylor, “Survey & Taxonomy of Packet Classification Techniques”, Technical Report, Washington University in Saint-Louis, USA, 2004.

[13] D. Liu, B. Hua, X. Hu and X. Tang, “High-performance Packet Classification Algorithm for Many-core and Multithreaded Network Processor”, Proc. of the 6th IEEE International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), 2006.

[14] P. Piyachon and Y. Luo, “Efficient Memory Utilization on Network Processors for Deep Packet Inspection”, ACM Symposium on Architectures for Network and Communications System (ANCS), 2006.

[15] L. Zhao, Y. Luo, L. Bhuyan and R. Iyer, “SpliceNP: A TCP Splicer using A Network Processor”, ACM Symposium on Architectures for Network and Communications System (ANCS), 2005.

[16] X. Hu, X. Tang and B. Hua, "High-Performance IPv6 Forwarding Algorithm for Multi-core and Multithreaded Network Processor". Proc. of ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming (PPoPP2006), 2006.

[17] Intel Corporation, “Intel IXP2850 Network Processor Hardware Reference Manual”, 2004.

[18] Intel Corporation, “Intel IXDP2850 Advanced Development Platform System User’s Manual”, 2004.

[19] B. Carlson, “Intel Internet Exchange Architecture and Applications”, Intel Press, 2003.

[20] E. J. Johnson and A. R. Kunze, “IXP2400/2850 Programming”, Intel Press, 2003.

[21] M. Venkatachalam, P. Chandra and R. Yavatkar, “A Highly Flexible, Distributed Multiprocessor Architecture for Network Processing”, Computer Networks, 2003.

[22] Y. Qi and J. Li, “Towards Effective Packet Classification”, Proc. of the Conference on Communication, Network, and Information Security (CNIS 2006), 2006.

[23] D. Srinivasan and W. Feng, “Performance Analysis of Multi-dimensional Packet Classification on Programmable Network Processors”, Proc. of the 29th Annual IEEE International Conference on Local Computer Networks (LCN), 2004.

[24] Intel Corporation, “Intel IXP2400 and IXP2800 Network Processor Programmer’s Reference Manual”, 2004.

[25] http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm

[26] S. Dharmapurikar and V. Paxson, “Robust TCP stream reassembly in the presence of adversaries”. Proc. of the 14th USENIX Security Symposium, 2005.

[27] G. Varghese, J. A. Fingerhut and F. Bonomi, “Detecting evasion attacks at high speeds without reassembly”, Proc. of ACM SIGCOMM, 2006.

[28] G. Varghese, “Network Algorithmics”, Elsevier Press, 2005.

[29] Y. Qi, B. Xu, F. He, X. Zhou, J. Yu and J. Li, “Towards Optimized Packet Classification Algorithms for Multi-Core Network Processors”, Proc. of the 2007 International Conference on Parallel Processing, 2007.

[30] http://www.arl.wustl.edu/~hs1/PClassEval.html

[31] Z. Cao, Z. Wang, E. Zegura, “Performance of Hashing-Based Schemes for Internet Load Balancing”, Proc. IEEE INFOCOM, 2000.

[32] W. Shi, M. H. MacGregor and P. Gburzynski, “A Scalable Load Balancer for Forwarding Internet Traffic: Exploiting Flow-level Burstiness”, ACM Symposium on Architectures for Network and Communications System (ANCS), 2005.

[33] T. L. Riché, J. Mudigonda, and H. M. Vin, “Experimental Evaluation of Load Balancers in Packet Processing Systems”, Proc. of the 1st Workshop on Building Block Engine Architectures for Computers and Networks (BEACON-1), 2004.

[34] http://pma.nlanr.net/Special/

[35] A. W. Moore and D. Zuev, "Internet Traffic Classification Using Bayesian Analysis Techniques," Proc. of the ACM SIGMETRICS, 2005.

[36] T. Karagiannis, K. Papagiannaki and M. Faloutsos, "BLINC: Multilevel traffic classification in the dark" (2005). Proc. of ACM SIGCOMM, 2005.

[37] S. Sen, O. Spatscheck and D. Wang, “Accurate, Scalable In-network Identification of p2p Traffic using Application Signatures”, Proc. of the 13th international Conference on World Wide Web, 2005.