TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 05/18 pp575-585
Volume 14, Number 5, October 2009

Efficiency of Cache Mechanism for Network Processors*

XU Bo (徐 波)[1,3], CHANG Jian (常 剑)[2], HUANG Shimeng (黄诗萌)[1],
XUE Yibo (薛一波)[3,4], LI Jun (李 军)[3,4,**]

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

Abstract: With the explosion of network bandwidth and the ever-changing requirements for diverse network-based applications, the traditional processing architectures, i.e., general purpose processor (GPP) and application specific integrated circuits (ASIC) cannot provide sufficient flexibility and high performance at the same time. Thus, the network processor (NP) has emerged as an alternative to meet these dual demands for today's network processing. The NP combines embedded multi-threaded cores with a rich memory hierarchy that can adapt to different networking circumstances when customized by the application developers. In today's NP architectures, multithreading prevails over cache mechanism, which has achieved great success in GPP to hide memory access latencies. This paper focuses on the efficiency of the cache mechanism in an NP. Theoretical timing models of packet processing are established for evaluating cache efficiency and experiments are performed based on real-life network backbone traces. Testing results show that an improvement of nearly 70% can be gained in throughput with assistance from the cache mechanism. Accordingly, the cache mechanism is still efficient and irreplaceable in network processing, despite the existing of multithreading.

Key words: cache; network processor; efficiency evaluation

Introduction

Due to the rapid increase of network bandwidth, general purpose processors (GPPs) are becoming incompetent to satisfy the requirements of OSI Layer 3 or Layer 4 network processing at line speeds of OC-192 (10 Gbps) and higher. Meanwhile, the ever-changing network environments and thousands of newly-emerging applications are gradually making application specific integrated circuits (ASICs) obsolete due to their long cycle and high cost of research and development. As a result, network processors (NPs) have become a promising alternative for high performance networking and security gateway design.

Since NPs balance flexibility and processing speed, they are expected to achieve both ASIC's high performance and GPP's time-to-market advantage, beneficial from their distributed, multiprocessor, multi-threaded architectures and the programming flexibility. Generally, a well-designed NP architecture should meet three requirements.

- **High processing speed** The Internet backbone has already reached the OC-192 line rate and is approaching the OC-768 rate (40 Gbps), which greatly shortens the processing time for each packet. Then NPs must be very fast to operate in real-time.
- **Ease of use** Industrial competition has caused the

Received: 2008-07-08; revised: 2009-05-13
* 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

product development cycle to be the decisive factor for system design companies to achieve fast time-to-market. Thus, NPs must enable very fast development time.

- **Flexibility** New protocols and applications are constantly emerging in the network community. Thus, NP programming and upgrading must be easy and flexible to extend the products' time-in-market.

Many integrated circuit companies, such as Intel[1], AMCC[2], Freescale[3], and Agere[4], have developed programmable NP products. Figure 1 shows the hardware architecture of the Intel IXP 2800 NP, one of the most popular high-end NP chips.

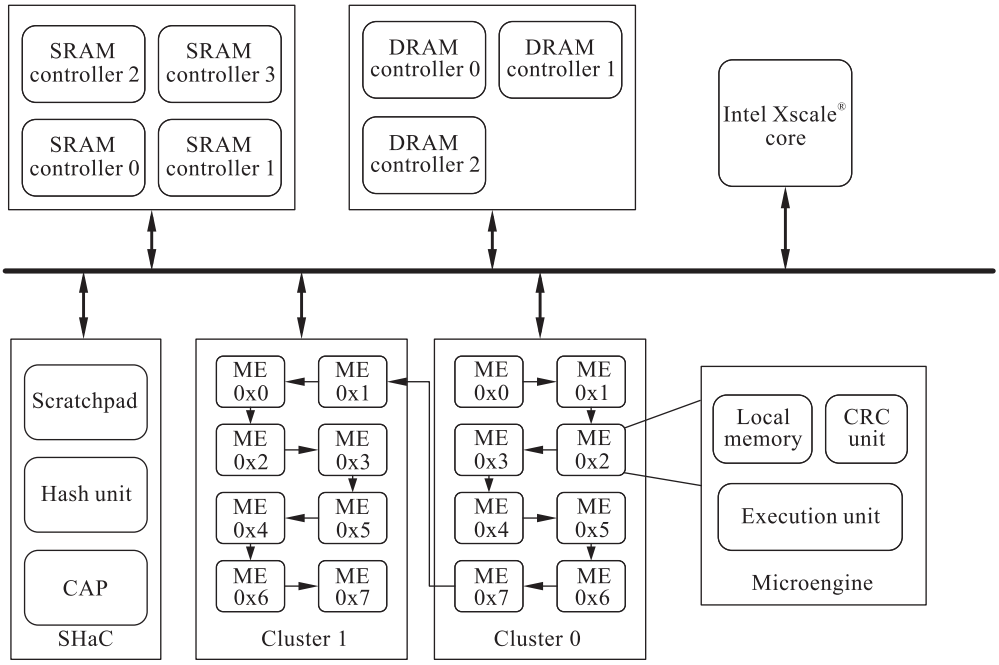

Fig. 1 Hardware architecture of Intel IXP 2800

As shown in Fig. 1, the IXP 2800 NP has two clusters of micro-engines (MEs), each of which supports 8 threads without context switching overhead. In addition, the IXP 2800 NP has a hierarchy of memory banks with various access speeds. The memory banks include registers, local memories (LMs) embedded in MEs, a scratchpad shared by all the MEs, and 4 channels of off-chip static random access memory (SRAM) banks along with 3 channels of off-chip dynamic random access memory (DRAM) banks.

In high-end NP architectures, the design of memory hierarchy is playing a more and more important role for the following reasons.

- The memory access speed is far behind the processing speed of the NP cores and the gap is increasing. Some NP cores can reach a working frequency of 1.4 GHz[1] while the off-chip SRAM needs 150 core cycles and DRAM needs 300 core cycles for one memory access. Hence, the memory access speed is the processing bottleneck and the most challenging aspect of NP designs.
- The memory hierarchy design greatly impacts the application development as well as the overall performance. An efficient flexible memory hierarchy greatly facilitates application development, which reduces the time-to-market and extends the product time.

Traditional computer architectures are typically comprised of three levels of memories: cache, main memory, and hard disk. These three memory levels conform to the inclusion principle, which means that data stored in the higher memory levels should also have copies in the lower levels. The inclusion principle facilitates the programming to a great extent. With cache mechanism and memory management unit (MMU), application developers do not have to explicitly control the data transfer between different levels, since the memory control is transparent to the programs.

However, unlike traditional architectures, the majority of today's NP products do not provide hardware-enhanced data transfer mechanisms such as cache. Instead, the memory hierarchy is directly revealed

to the programmers so the data structures must be explicitly allocated to the various memory banks. Besides, the data transfer must be controlled by instructions. Without employment of cache mechanism, many NP vendors are utilizing multithreading to improve the performance due to its advantage of hiding memory access latencies.

This paper analyzes the difference between multithreading and the cache mechanism and whether the cache mechanism can be an efficient addition to NP designs. This paper presents theoretical timing models for packet processing to evaluate cache efficiency in NPs. Experiments with real-life traces strongly support the viewpoint that the cache mechanism can achieve high efficiency in NP.

1 Background

This paper focuses on the efficiency of the cache mechanism in multi-core and multithreaded NPs. A brief introduction to the cache mechanism is provided first, followed by a comparison with multithreading. Then related work on the NP cache mechanism is discussed.

1.1 Cache mechanism principles

The cache discussed in this paper is assumed to be high speed memory with hardware assistance for maintenance and automatic updating. The cache mechanism is based on the locality principle, which means that every program will only access a small range of memory addresses within a short period of time. The locality includes temporal locality indicating that an entry is likely to be accessed again if it was just accessed and spatial locality indicating that nearby entries are likely to be accessed just after an entry is accessed.

Generally speaking, there are three types of memory latency hiding technologies, which are widely used nowadays: (a) prefetch mechanism which strives to avoid memory accesses; (b) cache mechanism which aims to shorten the memory access latencies; (c) multithreading mechanism which tries to hide the memory access latencies by simultaneous instruction executions.

Since the prefetch mechanism is difficult to implement on NPs, multithreading and the cache mechanism remain as the candidate schemes. Multithreading improves NP core utilization and delivers stable performance in spite of the data locality. However, each thread needs its own set of registers, which requires a large memory access bandwidth. The programming difficulty is another challenge. On the other side, the cache mechanism can shorten the waiting times of NP cores, and simplify the design of applications. However, the cache mechanism needs high speed memory and hardware assistance. Besides, it relies on data locality and suffers performance decline due to cache misses.

1.2 Previous work on NPs

There are two main concerns that have restricted the employment of the cache mechanism in NPs.

- First, network processing is assumed to lack data locality, both temporal and spatial. Hence, the cache hit rate will be quite low[5].
- Second, the average memory access time with cache mechanism is not constant due to cache misses, which conflicts with the real-time requirements of network processing. This might introduce jitter into the entire system performance[6].

In recent years, there have been some papers discussing cache mechanisms for NPs, mainly using them for routing table lookup[7-9] and flow classification[10-13] applications. However, these two prime concerns are not addressed directly in these papers.

The original attempt to evaluate cache efficiency in network processing was given by Memik et al.[14] They set up a benchmark named NetBench and used SimpleScalar[15] as the evaluation simulator. Actually, NetBench is designed for testing parameters related to instruction level parallelism, branch prediction accuracy, instruction distribution, and cache entry distribution. SimpleScalar is based on the general purpose processor Alpha 21264 with only one core, which is rather different from NP architectures. The shortcoming greatly weakened the applicability of the simulation results. Wolf and Franklin[16] conducted similar simulations on another benchmark named CommBench[16] but their analysis has the same problem.

Afterwards, Mudigonda et al.[17,18] improved the SimpleScalar simulator to support multithreading to evaluate the cache mechanism in NPs. They tested applications including flow classification and traffic

auditing with real-life rule sets to show that the cache mechanism is more efficient in reducing memory access time than wide-word access and scratchpad access.

Although the work of Mudigonda et al. provided more convincing simulation results, the simulator was still based on general purpose processors and they did not look into the effect of data locality in real-life network backbone traces. This paper investigates the effect of data locality in network processing and evaluates cache efficiency in NP architectures. Experiments with real-life Internet backbone traffic traces are conducted and the throughput speedup is calculated according to theoretical timing models.

2 Data Locality in Network Processing

This section analyzes the data locality in network processing procedures based on session lookup applications with the locality statistics from actual traces.

2.1 Data access mode in session lookup

Session lookup is such an important module in modern network processing that it is widely used in network devices including firewalls, intrusion detection/prevention systems, and many other network gateway devices. The function of session lookup is to search for the matching entry in a session table related to certain header fields of a packet. The session table is typically implemented as a hash table which records the states and actions of incoming and outgoing flows. Thus, packets belonging to the same flow can be associated to their corresponding processing actions directly without going through the tedious packet classification procedure used for the first flow packet.

The data locality in session lookup applications is analyzed based on the data access model. The session lookup procedure includes: (1) extract related packet header fields; (2) lookup the session table with the hash key generated from the packet header fields; (3) if a matching entry exists, update its states; otherwise, pass the packet to the packet classification module which is the first procedure in session creation.

The session lookup process is quite simple and there are hardware optimized instructions on NP to do the key operations such as fetching the packet header fields and computing hash values. The code is normally compact and can be placed in high speed instruction memory such as a control store or LM. For example, the IXP 2800 NP supports hardware-assisted cyclical redundancy check (CRC) hash and the session lookup program takes only 50 lines of microcode in our implementation. However, the session table data structure is often very large and can only be allocated to off-chip memory banks with lower access speeds, which introduces a potential bottleneck in the overall application performance.

Figure 2 shows the pseudo code for the primary memory accesses in session lookup.

```
Session_Lookup ( )
{
    Get_Packet_Header ();
    {
    Access on-chip local memory to get load instructions;
    Access off-chip DRAM to get packet header;
    }
    Session_Table_Access ();
    {
    Access on-chip local memory to get lookup instructions;
    Access off-chip DRAM to index session table;
    }
    Session_Table_Update ();
    {
    Access on-chip local memory to get update instructions;
    Write off-chip DRAM to update session table;
    }
}
```

Fig. 2 Data access mode in session lookup

According to the code, the processing time for a session lookup can be calculated as

$$T_{\text{Process}} = N_{\text{Instr}}(T_{\text{InstrExcute}} + T_{\text{InstrAccess}}) + T_{\text{PacketRAMAccess}} + (1+\beta)T_{\text{TableRAMAccess}} + T_{\text{TableRAMWrite}} \quad (1)$$

Here, N_{Instr} denotes the number of instruction lines. $T_{\text{InstrExcute}}$ represents the execution time for each instruction; $T_{\text{InstrAccess}}$ is the total memory access time for loading instructions; $T_{\text{PacketRAMAccess}}$ is the memory access time for fetching packet header fields; $T_{\text{TableRAMAccess}}$ is the time needed for one session lookup; $T_{\text{TableRAMWrite}}$ is the time needed for one session update; and β denotes the additional times of session table lookups when hash collisions occur. Session creation and teardown are not considered here since they have no relation with cache efficiency.

2.2 Data locality in session lookup

This analysis is based on the CRC hash as the hash function, which is a well-known procedure with hardware support by many NP chips. The session table size is set to 64K entries. In addition, this analysis focuses on the temporal locality, since the session table is precisely indexed by the hash value, which is unrelated to spatial locality.

For the processing time given in Eq. (1), N_{Instr} is determined by the complexity of the session lookup application, for example 50 lines of microcode on the IXP. $T_{\text{InstrExcute}}$ is determined by the instruction processing speed of the NP cores while $T_{\text{InstrAccess}}$ is determined by the control store access speed. Incoming packets are stored in the packet buffer in off-chip DRAM, which needs $T_{\text{PacketRAMAccess}}$ each time to get a packet header. The large session table is normally stored in off-chip DRAM memory banks.

Therefore, the target of the cache mechanism in session lookup applications is to reduce the data access time for each session table read and write, i.e., $T_{\text{TableRAMAccess}}$ and $T_{\text{TableRAMWrite}}$. The data locality of the session lookup application was then analyzed using a network backbone trace with this objective in mind.

The trace is of real-life traffic acquired from the OC-192 backbone of the Internet2's Indianapolis (IPLS) Abilene router node toward Kansas City (KSCY)[19-21], which is considered to be representative of Internet backbone traces. The trace contains a large number of flows with a stable packet arrival mode that is not heavily influenced by individual factors or burst accesses. The dataset has a total of 50M packets, which is equal to 10 min of traffic volume at IPLS.

To evaluate the dynamic data locality, traffic was analyzed in time intervals by treating each consecutive 50K packets as a segment. Hence, the entire traffic was split into 1000 time intervals of around 0.6 s. Analysis of the statistical characteristics of the time intervals is as follows:

- The number of matching session entries in each time interval is relatively stable, with an average value of 7658 for each 50K packets. Figure 3 shows the matching entry numbers for all the 1000 time intervals, which shows that the number of flows within each time interval is relatively stable. The data also implies that the CRC hash performance in the various intervals of the trace was quite steady.
- To illustrate the packet scattering in different flows, one time interval is picked as an example. Figure 4 depicts the hit times for the session entries in decreasing order while Fig. 5 shows the cumulative hit rate for the session entries. Figure 5 indicates that nearly 70% of the packets are from the most offen hit 1000 flows, which is around 1/7 of the total number of flows. This result confirms that the majority of the network traffic belongs to a small set of flows[22].

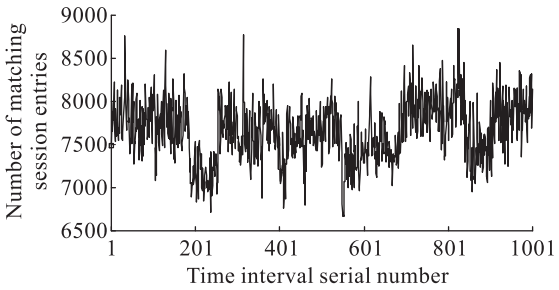

Fig. 3 Matching entry number

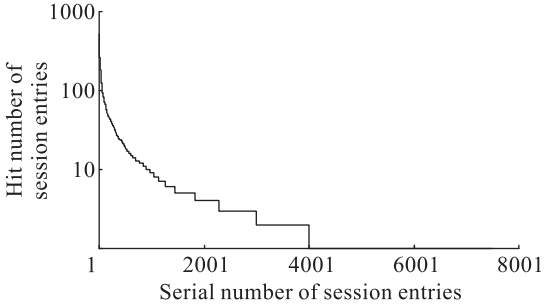

Fig. 4 Entry hit distribution

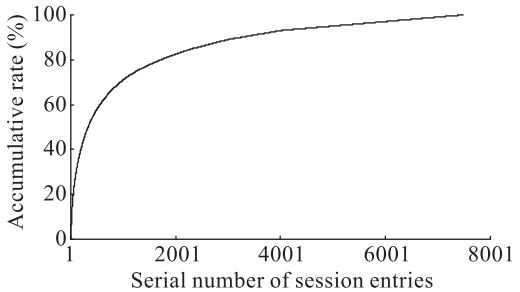

Fig. 5 Accumulative hit rate

- The data locality in session lookup was analyzed by counting the rate of session entries that are hit more than 10 times in each time interval as shown in Fig. 6. The most hit session entries show that around 12% are hit more than 10 times in each time interval. The curve shows that the rate does not vary much in different time intervals, indicating that the data locality is relatively stable.
- The number of packets belonging to the session

entries that were hit more than 10 times in each time interval was then counted to determine the fraction of the most frequent packets in the 50K packets for each time interval. Figure 7 shows that around 70% of the packets belong to the session entries which were hit more than 10 times. Figure 8 provides a more intuitive chart of the data locality in the session lookup application, with about 70% of the packets contained in about 12% of the flows.

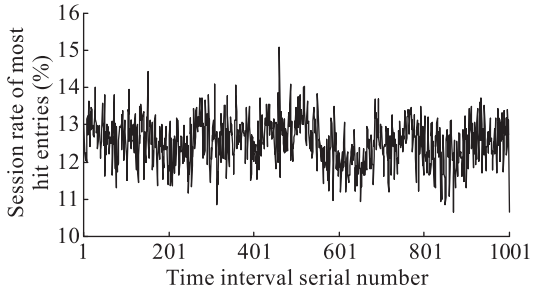

Fig. 6 Most hit session rate

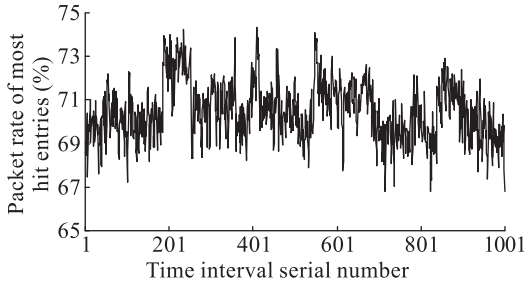

Fig. 7 Most hit packet rate

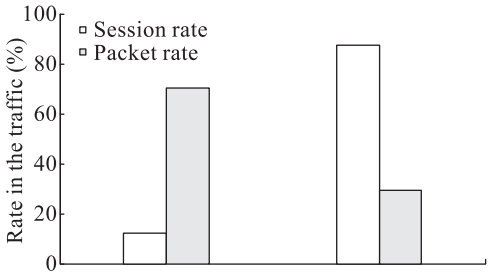

Fig. 8 Data locality

The observations based on the real-life trace show that the processing time can be greatly reduced if the most frequently hit entries are cached, which includes around 70% of the packets. These observations provide evidence of the data locality in network processing which motivates this work.

3 Timing Model of Cache Efficiency in NP

This section describes the timing model of session lookup application for the various cache mechanisms in NP and presents a criterion for cache efficiency. A multi-core NP timing model that supports multithreading is also presented. Fully set-associative cache is assumed here.

3.1 Criterion for cache efficiency

Three performance metrics are widely used in network processing: (1) the throughput which reflects the average processing power; (2) the packet loss ratio which reflects the peak value and the buffer bound of applications; (3) the queuing latency which reflects the burst processing ability. For all the three targets, the processing time for each packet is still the most crucial issue. Thus, the packet processing procedure can be simplified as in Fig. 9 to give the criterion for cache efficiency.

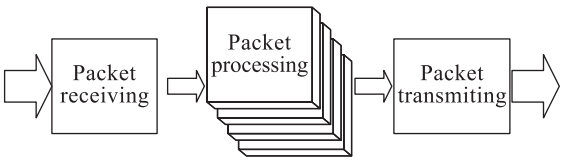

Fig. 9 Simple packet processing model in NP

The total time of single packet processing is given by

$$T_{\text{Total}} = T_{\text{PackRv}} + T_{\text{Process}} + T_{\text{PackTx}} \tag{2}$$

where T_{Process} consists of the instruction access time $T_{\text{InstrAccess}}$, the instruction execution time $T_{\text{InstrExcute}}$, and the data access time $T_{\text{DataAccess}}$; T_{PackRv} is the time needed for receiving a packet; and T_{PackTx} is the time needed for transmitting a packet. Since the cache mechanism only affects the data access time, the other times can be seen as constant values as shown in Eq. (3).

$$\begin{aligned} &T_{\text{Total}} = C + T_{\text{DataAccess}}, \\ &C = T_{\text{PackRv}} + T_{\text{InstrAccess}} + T_{\text{InstrExcute}} + T_{\text{PackTx}} \end{aligned} \tag{3}$$

The cache data access time is determined as Eq. (4).

$$T_{\text{DataAccess}} = \begin{cases} T_{\text{CacheAccess}}, & \text{when cache hit;} \\ T_{\text{CacheAccess}} + T_{\text{RAMAccess}}, & \text{when cache miss} \end{cases} \tag{4}$$

The average data access time is then given by Eq. (5), where $P_{\text{CacheMiss}}$ is the probability of cache miss.

$$T_{\text{AVGDataAccess}} = T_{\text{CacheAccess}} + P_{\text{CacheMiss}} T_{\text{RAMAccess}} \tag{5}$$

Since $P_{\text{CacheMiss}}$ is a statistical probability, which is closely related to the hardware implementation and the cache replacement algorithm, the effect of the precise cache miss rate for real-life circumstances cannot be

analyzed. Thereby, a cache jitter rate is used to represent the cache miss characteristics. If ΔC_{Cache} is used to denote the swap entries in the cache during a time slice Δt, the cache jitter rate $R_{\text{CacheJitter}}$ can be calculated by

$$R_{\text{CacheJitter}} = \frac{\Delta C_{\text{Cache}}}{C_{\text{Cache}}} \tag{6}$$

where C_{Cache} is the total capacity of the cache table, typically represented by the number of entries. If Δt is small enough, a good mathematical approximation can be denoted as Eq. (7). Therefore, the cache mechanism can be evaluated by comparing the data access times with and without a cache.

$$R_{\text{CacheJitter}} \approx P_{\text{CacheMiss}} \tag{7}$$

Equation (8) presents the theoretical criterion for cache efficiency. The cache mechanism is useful when the average memory access time is less than the access time without a cache. The efficiency will have different ranges for the various processing architectures as well as the various hash functions.

$$T_{\text{RAMAccess}} \geqslant T_{\text{CacheAccess}} + R_{\text{CacheJitter}} T_{\text{RAMAccess}} \tag{8}$$

3.2 Timing models with a cache on an NP

This section describes the timing models for three kinds of cache mechanisms in NPs. The shared cache model is the most commonly used. The distributed cache model designates a stand-alone cache for each core, while the extended shared cache model supports multithreading.

3.2.1 Shared cache model

The classic cache modes include fully associated cache (FAC), set associated cache (SAC), and direct mapped cache (DMC). Cache updating algorithms include least recently used (LRU), most recently used (MRU), least frequently used (LFU), and adaptive replacement cache (ARC). This simulation uses the FAC mode for the cache model and the LFU algorithm for ease of interpretation.

As mentioned in Section 2.2, the cache jitter rate is used to approximate the cache miss rate. Figure 10 shows the number of cache entry swaps in each time interval for different cache table sizes while Fig. 11 depicts the cache jitter rate in one time interval for different cache table sizes. As shown, the cache jitter rate gradually decreases as the cache table size increases, with the jitter rate reduced to around 30% for a cache size of 1000 entries.

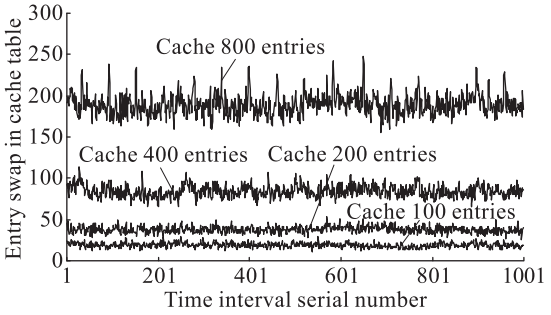

Fig. 10 Entry swap in cache table

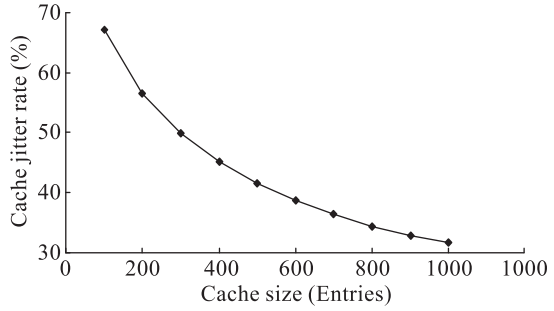

Fig. 11 Cache table jitter rate

For this cache mechanism, the packet processing time in Eq. (1) can be written as

$$T_{\text{ProcessCache}} = N_{\text{Instr}}(T_{\text{InstrExcute}} + T_{\text{InstrAccess}}) + T_{\text{PacketRAMAccess}} + T_{\text{CacheWrite}} + (1+\beta)[(1-R_{\text{CacheJitter}})T_{\text{CacheAccess}} + R_{\text{CacheJitter}}(T_{\text{CacheAccess}} + T_{\text{TableRAMAccess}})] \tag{9}$$

3.2.2 Distributed cache model

Figure 12 shows the distributed cache architecture in an NP, which assumes that the packets are distributed into the cores at the granularity of flows to facilitate the session lookup. The packet processing time can still be expressed as Eq. (9).

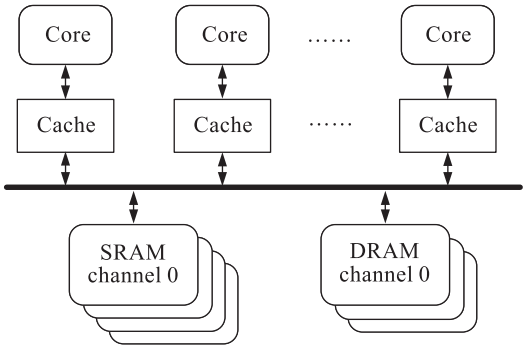

Fig. 12 Distributed cache model in NP

The trace was split into 16 sub-traces with Fig. 13 giving the average jitter rate for the 16 sub-traces with different cache sizes. The result is rather similar to that of the shared cache model shown in Fig. 11, because

the flows are scattered evenly into the 16 sub-traces as shown in Fig. 14. Here, the cumulative hit rates of the session entries within the 16 sub-traces have similar distributions as the original shared cache model illustrated in Fig. 5.

3.2.3 Shared cache model with multithreading

Multithreading is used by most NP vendors to compensate for the lack of cache. The cache mechanism reduces the memory access time while multithreading seeks to hide the memory access latencies when they cannot be effectively reduced. Although multithreading can reduce the access latencies when the cache is not efficient[18], the cache should not be completely replaced by multithreading due to the locality described earlier. In addition, multithreading will bring additional waiting time because of the mutual exclusion lock between threads. Consequently, the processing time using both cache and multithreading can be denoted as

$$T_{\text{ProcessCacheMultithread}}=T_{\text{Mutex}}+\text{Max}\{N_{\text{Instr}}(T_{\text{InstrExcute}}+T_{\text{InstrAccess}}),\ T_{\text{PacketRAMAccess}}+(1+\beta)[(1-R_{\text{CacheJitter}})T_{\text{CacheAccess}}+R_{\text{CacheJitter}}\cdot (T_{\text{CacheAccess}}+T_{\text{TableRAMAccess}})]+T_{\text{CacheWrite}}\} \quad (10)$$

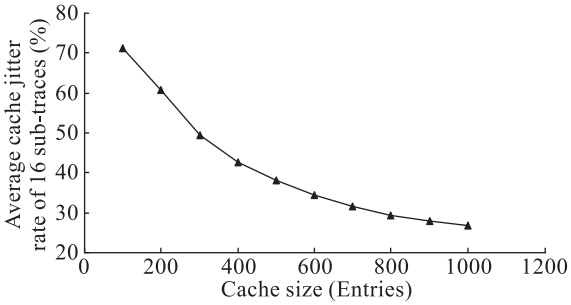

Fig. 13 Average cache table jitter

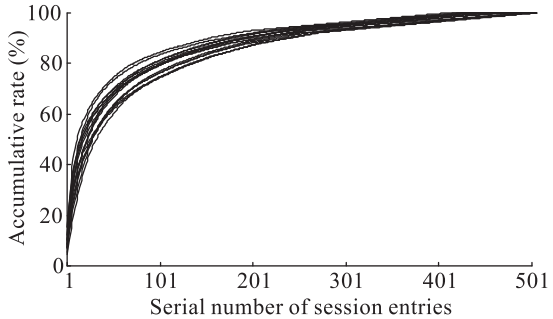

Fig. 14 Accumulative hit rate

4 Performance Evaluation

To evaluate the cache efficiency in an NP, the IXP 2800 is selected as the testing architecture and the simulation results are obtained based on the real-life OC-192 backbone trace provided by IPLS[19-21]. The CRC hash is adopted here with experiments conducted to evaluate the cache miss rate and the throughput speedups with the various cache mechanisms in NP. The worst-case performance is evaluated to analyze the influence on throughput and packet processing jitter rate.

Since there is no actual cache hardware on an NP, the simulation used the cache jitter rate to approximate the cache miss rate. As mentioned in Section 3.1, if the time slice Δt is small enough, the cache miss rate can be estimated as Eq. (7).

4.1 Cache miss rate

Since multithreading has no influence on the cache miss rate with flow granularity splitting, only the shared cache model was compared with the distributed cache model. The simulations calculated the cache jitter rates for short time period intervals with their average value used as the cache miss rate as an approximation.

Figure 15 illustrates the cache miss rates for various cache table sizes, which shows that the cache miss rate of session lookup application with real-life network trace can decrease to 25%-30% with a cache size of 1000 entries. The cache miss rate decreases gradually as the cache size increases, but the reducing speed is slowing down when the cache size grows. The reason is that the backbone network trace has an uneven distribution with 70% of the packets belonging to only 12% of the flows, as shown in Fig. 8. This means that only 12% of the session entries should be cached. If the cache table size is larger than this amount, the cache table has to store more entries that are not frequently accessed. Thus, the cache miss rate can hardly be reduced via the increasing of cache size.

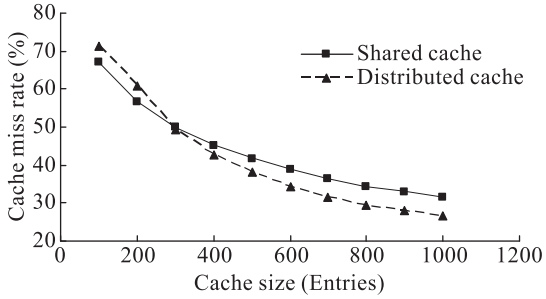

Fig. 15 Cache miss rate

To verify this reasoning, additional tests with larger cache table sizes were performed and the trend of the cache miss rate with increasing cache table sizes is described in Fig. 16. It shows that the cache miss rate decreases with the cache table size growing. But the increasing speed slows down when the cache table size exceeds 1000 entries.

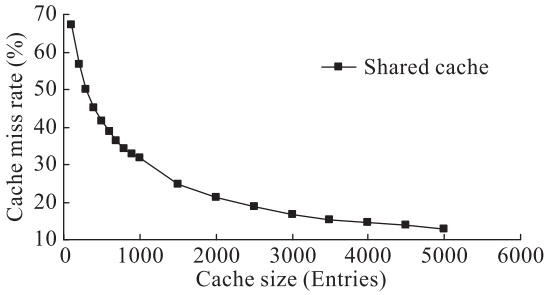

Fig. 16 Trend of cache miss rate

Another observation from Fig. 15 is that the cache miss rate of the distributed cache model is very close to that of the shared cache model. This is because the trace is evenly spit into 16 sub-traces by the CRC hash at the flow level granularity. Thus, the data locality characteristics of the 16 sub-traces are similar to the original backbone trace.

4.2 Throughput speedup

The throughput speedup of the three cache models was also analyzed based on the IXP 2800 NP. Table 1[1] lists the read and write times needed for each memory bank on the IXP 2800 architecture.

Table 1 Access time of memory banks on IXP 2800

Memory banks	Volume	Read (cycles)	Write (cycles)
Local memory	640*4	5	5
Scratchpad	16 K	100	40
SRAM	64 M	130	53
DRAM	2 G	295	53

The instructions are normally stored in LM and the analysis assumes that the session lookup table is stored in DRAM because of the enormous size. The throughput speedup with a cache mechanism was computed as Eq. (11) and the processing time with a cache can be estimated from Eqs. (9) and (10). Both the shared cache model and the distributed cache model comply with Eq. (9) but they have different cache miss rates, while the shared cache model and the extended shared cache model with multithreading have the same cache miss rate but have different timing models.

$$\text{Speedup} = T_{\text{Process}} / T_{\text{ProcessCache}} \tag{11}$$

If the cache has the same read and write speeds as the in-chip LM, the cache access times $T_{\text{CacheAccess}}$ and $T_{\text{CacheWrite}}$ will be 5 cycles. $T_{\text{InstrExcute}} + T_{\text{InstrAccess}}$ are most likely to be 1 cycle if the NP uses instruction pipelining. The session lookup program for the IXP 2800 is about 50 lines of microcode. β is calculated as $\beta = \alpha(L-1)/2$, where α is the hash collision rate and L is the depth of the session hash link list. When the hash load factor is set to 1/2, the hash collision rate α is 10.68% and the longest hash link list is 6 based on tests with the CRC hash. $T_{\text{PacketRAMAccess}}$ is 295 cycles if the packet buffer is in DRAM. $T_{\text{TableRAMAccess}}$ is 295 cycles and $T_{\text{TableRAMWrite}}$ is 53 cycles if the session table is stored in DRAM. T_{Mutex} is estimated to be 50 cycles. The throughput speedup for the various cache models was then estimated for different cache sizes.

Figure 17 shows the throughput speedup for the three cache models. Up to 65%-70% of performance gains in throughput can be obtained by introducing a cache mechanism into the IXP 2800 NP platform. The speedup with the shared cache model can reach 65% while the speedup with the distributed cache model can reach 70%. The speedup disparity is due to the different cache miss rates of the two models.

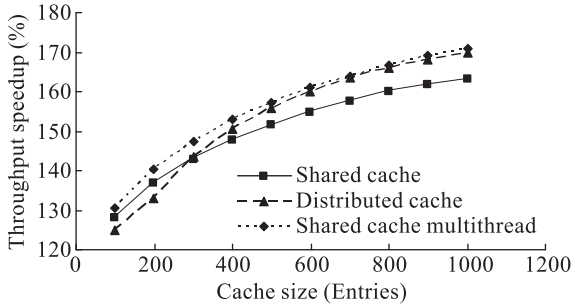

Fig. 17 Throughput speedup

Another result is that the throughput speedup of the extended shared cache model with multithreading has a higher performance gain of up to 70% compared with the shared cache model. Thus, the multithreading introduces a performance gain of 5%-8%. It provides convincing evidence that the cache mechanism has a remarkable effect on NPs even with multithreading. This effect is attributed to the different functions of the two mechanisms. The cache mechanism reduces the memory access time while multithreading hides some

of the instruction execution time by the memory accesses. Therefore, the two mechanisms can collaborate with each other to achieve higher improvement in the overall network processing speed.

In addition, the speedup of all three models increases sub-linearly with the increasing of cache size, because the cache miss rate declines gradually as the cache size grows, resulting in lower average memory access time.

4.3 Worst-case evaluation

The memory access time with the cache mechanism is not constant due to cache misses, which introduce jitter into the packet processing speed. The worst-case condition assumes that the cache miss rate is 100%. Thus, the processing time will be $T_{\text{CacheAccess}} + T_{\text{RAMAccess}}$ for each packet. Compared with the $T_{\text{RAMAccess}}$ time for each packet without a cache, the worst case only introduces a slight overhead of $T_{\text{CacheAccess}}$, which is normally much smaller than $T_{\text{RAMAccess}}$. For the access times in Table 1, the worst-case throughput is illustrated in Fig. 18, which shows that the worst-case performance is still nearly 99% of the processing speed without the cache mechanism.

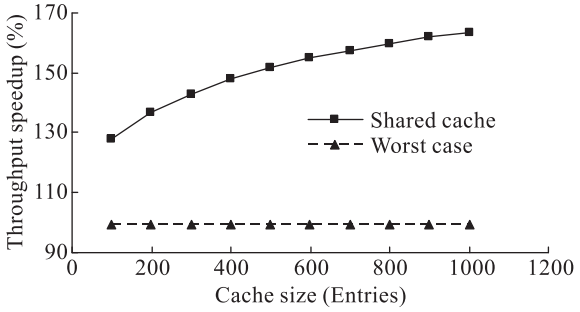

Fig. 18 Worst-case performance

Figure 18 also indicates that compared with the average throughput for the shared cache model, the worst-case situation causes throughput reduction of 22%-40%, which will bring about jitter in the packet processing speed, which conflicts with the real-time requirement of network processing. However, the jitter can be reduced by larger packet buffers, since the cache misses will not last long in real traffic according to the previous observations.

5 Conclusions and Future Work

This paper analyzes the efficiency of the cache mechanism in NPs with a criterion for evaluating the cache efficiency. Theoretical timing models for three typical cache mechanisms were described based on the Intel IXP 2800 NP with a real-life network backbone trace. The cache miss rates were estimated using the cache jitter rates with the throughput speedups evaluated according to the packet processing models.

Tests show that the cache miss rate of session lookup application can be reduced to 25%-30% with appropriate cache table sizes with throughput speedups of 65%-70%. It is proved that the cache mechanism is still quite effective in NP. Furthermore, the cache mechanism can collaborate with multithreading to achieve even higher performance speedups. The results demonstrate that cache is an efficient, irreplaceable part of NP cores, even with multithreading.

Future work includes detailed analyses of the mutual influences of cache and multithreading mechanisms. The hardware design of NP chips to embed considerable sized caches is another challenge, since cache chips are usually relatively large and the space on actual chips is quite limited.

Acknowledgements

The authors thank Qi Yaxuan, Zhou Guangyu, and all the other colleagues in the Network Security Lab for their suggestions and help.

References

[1] Intel. IXP2XXX product line of network processor. http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm, 2009.

[2] AMCC. Network processor. https://www.amcc.com/MyAMCC/jsp/public/browse/controller.jsp?networkLevel=COMM&superFamily=NETP, 2009.

[3] Freescale. C-Port network processors. http://www.freescale.com/webapp/sps/site/homepage.jsp?nodeId=02VS01DFTQ3126, 2009.

[4] Agere. Network processor. http://www.agere.com/telecom/network_processors.html, 2009.

[5] Venkatachalam M, Chandra P, Yavatkar R. A highly flexible, distributed multiprocessor architecture for network processing. *Computer Networks*, 2003, **41**(5): 563-586.

[6] Derek C, Prabhat J, Srinivas D, et al. Application-specific memory management for embedded systems using software-controlled caches. In: Proc. of the 37th Design Automation Conference (DAC). Los Angeles, CA, USA,

2000.

[7] Liu H. Routing prefix caching in network processor design. In: Proc. of the 10th International Conference on Computer Communications and Networks. Scottsdale, Arizona, USA, 2001.

[8] Gopalan K, Chiueh T. Improving route lookup performance using network processor cache. In: Proc. of the 5th International Symposium on High Performance Computer Architecture. Orlando, Florida, USA, 1999.

[9] Rajan K, Govindarajan R. A heterogeneously segmented cache architecture for a packet forwarding engine. In: Proc. of International Conference on Supercomputing (ICS). Cambridge, MA, USA, 2005.

[10] Xu J, Singhal M, Degroat J. A novel cache architecture to support layer-four packet classification at memory access speeds. In: Proc. of IEEE INFOCOM. Tel-Aviv, Israel, 2000.

[11] Tung Y, Che H. A flow caching mechanism for fast packet forwarding. *Computer Communications*, 2002, **25**(14): 1257-1262.

[12] Li K, Chang F, Berger D, et al. Architectures for packet classification caching. In: Proc. of the 11th IEEE International Conference on Networks. Sydney, Australia, 2003.

[13] Li B, Venkatesh G, Calder B, et al. Exploiting a computation reuse cache to reduce energy in network processors. In: Proc. of International Conference on High Performance Embedded Architectures and Compilers. Barcelona, Spain, 2005.

[14] Memik G, Mangione-Smith W H, Hu W. NetBench: A benchmarking suite for network processors. In: Proc. of IEEE/ACM International Conference on Computer-Aided Design. Braunschweig, Germany, 2001.

[15] http://www.simplescalar.com/, 2009.

[16] Wolf T, Franklin M. CommBench-a telecommunications benchmark for network processors. In: Proc. of IEEE International Symposium on Performance Analysis of Systems and Software. Austin, Texas, USA, 2000.

[17] Mudigonda J, Vin H M, Yavatkar R. Overcoming the memory wall in packet processing: hammers or ladder? In: Proc. of the Symposium on Architecture for Networking and Communications Systems. Princeton, NJ, USA, 2005.

[18] Mudigonda J, Vin H M, Yavatkar R. Managing memory access latency in packet processing. In: Proc. of the International Conference on Measurement and Modeling of Computer Systems. Banff, Canada, 2005.

[19] Abilene-III Trace data. http://pma.nlanr.net/Special/ipls3.html, 2009.

[20] Internet2. http://www.internet2.edu/, 2009.

[21] Abilene backbone network. http://abilene.internet2.edu/, 2009.

[22] Hamed H, El-Atawy A, Al-Shaer E. Adaptive statistical optimization techniques for firewall packet filtering. In: Proc. of IEEE INFOCOM. Barcelona, Spain, 2006.